

QCDNUM: Fast QCD Evolution and Convolution

QCDNUM Version 17.00

M. Botje*

Nikhef, Science Park, Amsterdam, the Netherlands

May 8, 2010

(Revised October 6, 2010)

Abstract

The QCDNUM program numerically solves the evolution equations for parton densities and fragmentation functions in perturbative QCD. Un-polarised parton densities can be evolved up to next-to-next-to-leading order in powers of the strong coupling constant, while polarised densities or fragmentation functions can be evolved up to next-to-leading order. Other types of evolution can be accessed by feeding alternative sets of evolution kernels into the program. A versatile convolution engine provides tools to compute parton luminosities, cross-sections in hadron-hadron scattering, and deep inelastic structure functions in the zero-mass scheme or in generalised mass schemes. Input to these calculations are either the QCDNUM evolved densities, or those read in from an external parton density repository. Included in the software distribution are packages to calculate zero-mass structure functions in un-polarised deep inelastic scattering, and heavy flavour contributions to these structure functions in the fixed flavour number scheme.

*Nikhef, Science Park 105, 1098XG Amsterdam, the Netherlands; email m.botje@nikhef.nl

PROGRAM SUMMARY

Program Title: QCDNUM

Version: 17.00

Author: M. Botje

E-mail: m.botje@nikhef.nl

Program obtainable from: <http://www.nikhef.nl/user/h24/qcdnum>

Distribution format: gzipped tar file

Journal Reference:

Catalogue identifier:

Licensing provisions: GNU Public License

Programming language: FORTRAN-77

Computer: all

Operating system: all

RAM: Typically 3 Mbytes

Keywords: QCD evolution, DGLAP evolution equations, Parton densities, Fragmentation functions, Structure functions

Classification: 11.5 Quantum Chromodynamics, Lattice Gauge Theory

External routines/libraries: none, except the MBUTIL, ZMSTF and HQSTF packages that are part of the QCDNUM software distribution.

Nature of problem: Evolution of the strong coupling constant and parton densities, up to next-to-next-to-leading order in perturbative QCD. Computation of observable quantities by Mellin convolution of the evolved densities with partonic cross-sections.

Solution method: Parametrisation of the parton densities as linear or quadratic splines on a discrete grid, and evolution of the spline coefficients by solving (coupled) triangular matrix equations with a forward substitution algorithm. Fast computation of convolution integrals as weighted sums of spline coefficients, with weights derived from user-given convolution kernels.

Restrictions: Accuracy and speed are determined by the density of the evolution grid.

Running time: Less than 10 ms on a 2 GHz Intel Core 2 Duo processor to evolve the gluon density and 12 quark densities at next-to-next-to-leading order over a large kinematic range.

Contents

1	Introduction	5
2	QCD Evolution	6
2.1	Evolution of the Strong Coupling Constant	6
2.2	The DGLAP Evolution Equations	7
2.3	Renormalisation Scale Dependence	10
2.4	Decomposition into Singlet and Non-singlets	10
2.5	Flavour Number Schemes	12
3	Numerical Method	14
3.1	Polynomial Spline Interpolation	15
3.2	Convolution Integrals	18
3.3	DGLAP Evolution	20
4	The QCDNUM Program	24
4.1	Source Code	24
4.2	Application Program	24
4.3	Validation and Performance	27
5	Subroutine Calls	29
5.1	Initialisation	31
5.2	Grid Definition	32
5.3	Weights	35
5.4	Parameters	36
5.5	Evolution	37
5.6	External Pdfs	39
5.7	Pdf Interpolation	40
6	Convolution Engine	42
6.1	Rescaling Variable in Convolution Integrals	42
6.2	Weight Tables	43
6.3	Convolution	50
6.4	Interpolation	52
6.5	Fast Computation	53

6.6	Custom Evolution	58
6.7	Error Messages in Add-On Packages	61
7	Acknowledgements	61
A	Singularities	62
B	Triangular Systems in the DGLAP Evolution	63
C	Zero Mass Structure Functions	64
C.1	General Formalism	64
C.2	Renormalisation and Factorisation Scale Dependence	65
C.3	The ZMSTF Package	66
D	Heavy Quark Structure Functions	68
D.1	The HQSTF Package	69
	References	71
	Index	73

1 Introduction

In perturbative quantum chromodynamics (pQCD), a hard hadron-hadron scattering cross section is calculated as the convolution of a partonic cross section with the momentum distributions of the partons inside the colliding hadrons. These parton distributions depend on the Bjorken- x variable (fractional momentum of the partons inside the hadron) and on a scale μ^2 characteristic of the hard scattering process. Whereas the x -dependence of the parton densities is non-perturbative, the μ^2 dependence can be described in pQCD by the DGLAP evolution equations [1]. The perturbative expansion of the splitting functions in these equations has recently been calculated up to next-to-next-to-leading order (NNLO) in powers of the strong coupling constant α_s [2, 3].

QCDNUM is a FORTRAN program that numerically solves the DGLAP evolution equations on a discrete grid in x and μ^2 . Input to the evolution are the x -dependence of the parton densities at some input mass factorisation scale, and an input value of α_s at some input renormalisation scale. To study the scale uncertainties, the renormalisation scale can be varied with respect to the mass factorisation scale. All calculations in QCDNUM are performed in the $\overline{\text{MS}}$ scheme.

The program was originally developed in 1988 by members of the BCDMS collaboration [4] for a next-to-leading order (NLO) pQCD analysis of the SLAC and BCDMS structure function data [5]. This code was adapted by the NMC for use at low x [6]. A complete revision led to the version 16.12 which was used in the QCD fits by ZEUS [7], and in a global QCD analysis of deep inelastic scattering data by the present author [8].

QCDNUM17 is the NNLO upgrade of QCDNUM16. A new evolution algorithm, based on quadratic spline interpolation, yields large gains in accuracy and speed; on a 2 GHz processor it takes less than 10 ms to evolve over a large kinematic range the full set of parton densities at NNLO in the variable flavour number scheme. QCDNUM17 can evolve un-polarised parton densities up to NNLO, and polarised densities or fragmentation functions up to NLO. Alternative sets of evolution kernels can be fed into the program to perform other types of evolution. It is also possible to read a parton density set from an external library, instead of evolving these from the input scale.

A versatile set of convolution routines is provided that can be used to calculate hadron-hadron scattering cross-sections, parton luminosities, or deep inelastic structure functions in either the zero-mass or in generalised mass schemes. Included in the software distribution are the ZMSTF and HQSTF add-on packages to compute un-polarised zero-mass structure functions and, in the fixed flavour number scheme, the contribution from heavy quarks to these structure functions.

This write-up is organised as follows. In Section 2 we summarise the formalism underlying the DGLAP evolution of parton densities. The QCDNUM numerical method is described in Section 3. Details about the program itself and the description of an example job can be found in Section 4. A subroutine-by-subroutine manual is given in Section 5. The QCDNUM convolution engine is presented in Section 6. The ZMSTF and HQSTF packages are described in the Appendices C and D, respectively.

2 QCD Evolution

In pQCD, the strong coupling constant α_s evolves on the renormalisation scale μ_R^2 . The starting value is specified at some input scale, which usually is taken to be m_Z^2 .

The parton density functions (pdf) evolve on the factorisation scale μ_F^2 . The starting point of a pdf evolution is given by the x dependence of the pdf at some initial scale μ_0^2 . The coupled evolution equations that are obeyed by the gluon and the quark densities can, to a large extent, be decoupled by writing them in terms of the *singlet* quark density (sum of all active quarks and anti-quarks) and *non-singlet* densities (orthogonal to the singlet in flavour space). A nice feature of QCDNUM is that it automatically takes care of the singlet/non-singlet decomposition of a set of pdfs.

Another input to the QCD evolution is the number of active flavours n_f which specifies how many quark species (d, u, s, ...) are participating in the QCD dynamics. In the *fixed flavour number scheme* (FFNS), n_f is kept fixed throughout the evolution. Input to an FFNS evolution are then the gluon density and $2n_f$ (anti-)quark densities at the input scale μ_0^2 . In the *variable flavour number scheme* (VFNS), the flavour thresholds $\mu_{c,b,t}^2$ are introduced and 3 light quark densities (d, u, s) are, together with the corresponding anti-quark densities, specified below the charm threshold μ_c^2 . The heavy quarks and anti-quarks (c, b, t) are dynamically generated by the QCD evolution equations at and above their thresholds. Both the FFNS and the VFNS are supported by QCDNUM.

The QCD evolution formalism is relatively simple when the renormalisation and factorisation scales are equal, but it becomes more complicated when $\mu_R^2 \neq \mu_F^2$. QCDNUM supports a linear relationship between the two scales.

In the following sections we describe the evolution of α_s and the pdfs, the renormalisation scale dependence, the singlet/non-singlet decomposition, and the flavour schemes.

2.1 Evolution of the Strong Coupling Constant

The evolution of the strong coupling constant reads, up to NNLO,

$$\frac{da_s(\mu^2)}{d \ln \mu^2} = - \sum_{i=0}^2 \beta_i a_s^{i+2}(\mu^2). \quad (2.1)$$

Here $\mu^2 = \mu_R^2$ is the renormalisation scale and $a_s = \alpha_s/2\pi$. The β -functions in (2.1) depend on the number n_f of active quarks with pole mass $m < \mu$. In the $\overline{\text{MS}}$ scheme they are given by [9, 10]

$$\begin{aligned} \beta_0 &= \frac{11}{2} - \frac{1}{3} n_f \\ \beta_1 &= \frac{51}{2} - \frac{19}{6} n_f \\ \beta_2 &= \frac{2857}{16} - \frac{5033}{144} n_f + \frac{325}{432} n_f^2. \end{aligned} \quad (2.2)$$

The leading order (LO) analytical solution of (2.1) can be written as

$$\frac{1}{a_s(\mu^2)} = \frac{1}{a_s(\mu_0^2)} + \beta_0 \ln \left(\frac{\mu^2}{\mu_0^2} \right) \equiv \beta_0 \ln \left(\frac{\mu^2}{\Lambda^2} \right). \quad (2.3)$$

In (2.3), the parameter Λ is defined as the scale where the first term on the right-hand side vanishes, that is, the scale where α_s becomes infinite. Beyond LO, the definition of a scale parameter is ambiguous so that it is more convenient to take $\alpha_s(m_Z^2)$ as a reference. The value of α_s at any other scale is then obtained from a numerical integration of (2.1),¹ instead of from approximate analytical solutions parametrised in terms of Λ .

In the evolution of α_s , the number of active flavours is set to $n_f = 3$ below the charm threshold $\mu_R^2 = \mu_c^2$ and is changed from n_f to $n_f + 1$ at the flavour thresholds $\mu_R^2 = \mu_{c,b,t}^2$. At NNLO, and sometimes also at NLO, there are small discontinuities in the α_s evolution at the flavour thresholds [11]; see Section 2.5 for details.

In Figure 1, we plot the evolution of α_s calculated at LO, NLO and NNLO.² Because

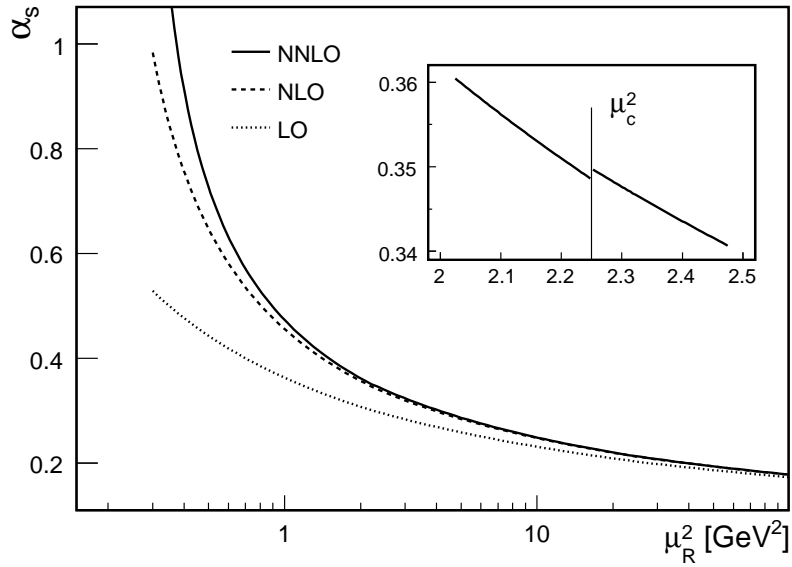


Figure 1: The strong coupling constant $\alpha_s(\mu_R^2)$ evolved downward from $\alpha_s(m_Z^2) = 0.118$ in LO (dotted curve), NLO (dashed curve) and NNLO (full curve). The inset shows an enlarged view of the NNLO discontinuity in α_s at the charm threshold μ_c^2 .

pQCD breaks down when α_s becomes large, QCDNUM will issue a fatal error when $\alpha_s(\mu^2)$ exceeds a pre-set limit. For a given value of $\alpha_s(m_Z^2)$, it is clear from the figure that such a limit will correspond to larger values of μ^2 at larger perturbative order.

2.2 The DGLAP Evolution Equations

The DGLAP evolution equations can be written as

$$\frac{\partial f_i(x, \mu^2)}{\partial \ln \mu^2} = \sum_{j=q, \bar{q}, g} \int_x^1 \frac{dz}{z} P_{ij} \left(\frac{x}{z}, \mu^2 \right) f_j(z, \mu^2) \quad (2.4)$$

¹I thank A. Vogt for providing his 4th order Runge-Kutta routine to integrate (2.1) up to NNLO.

²With the settings $\alpha_s(m_Z^2) = 0.118$ and $\mu_{c,b,t} = (1.5, 5, 188)$ GeV.

where f_i denotes an un-polarised parton number density, P_{ij} are the QCD splitting functions, x is the Bjorken scaling variable and $\mu^2 = \mu_F^2$ is the mass factorisation scale, which we assume here to be equal to the renormalisation scale μ_R^2 . The indices i and j in (2.4) run over the parton species *i.e.*, the gluon and n_f active flavours of quarks and anti-quarks. In the quark parton model, and also in LO pQCD, the parton densities are defined such that $f(x, \mu^2)dx$ is, at a given μ^2 , the number of partons which carry a fraction of the nucleon momentum between x and $x + dx$. The distribution $xf(x, \mu^2)$ is then the parton momentum density.³ Beyond LO there is no such intuitive interpretation. The definition of f then depends on the renormalisation and factorisation scheme in which the calculations are carried out ($\overline{\text{MS}}$ in QCDNUM).⁴

Introducing a short-hand notation for the Mellin convolution,

$$[f \otimes g](x) = \int_x^1 \frac{dz}{z} f\left(\frac{x}{z}\right) g(z) = \int_x^1 \frac{dz}{z} f(z) g\left(\frac{x}{z}\right), \quad (2.5)$$

we can write (2.4) in compact form as (we drop the arguments x and μ^2 in the following)

$$\frac{\partial f_i}{\partial \ln \mu^2} = \sum_{j=q, \bar{q}, g} P_{ij} \otimes f_j. \quad (2.6)$$

If the x dependencies of the parton densities are known at some scale μ_0^2 , they can be evolved to other values of μ^2 by solving this set of $2n_f + 1$ coupled integro-differential equations. Fortunately, (2.6) can be considerably simplified by taking the symmetries in the splitting functions into account [9]:

$$\begin{aligned} P_{gq_i} &= P_{g\bar{q}_i} = P_{gq} \\ P_{q_i g} &= P_{\bar{q}_i g} = \frac{1}{2n_f} P_{qg} \\ P_{q_i q_k} &= P_{\bar{q}_i \bar{q}_k} = \delta_{ik} P_{qq}^v + P_{qq}^s \\ P_{q_i \bar{q}_k} &= P_{\bar{q}_i q_k} = \delta_{ik} P_{q\bar{q}}^v + P_{q\bar{q}}^s. \end{aligned} \quad (2.7)$$

Inserting (2.7) in (2.6), we find after some algebra that the singlet quark density

$$q_s = \sum_{i=1}^{n_f} (q_i + \bar{q}_i) \quad (2.8)$$

obeys an evolution equation coupled to the gluon density

$$\frac{\partial}{\partial \ln \mu^2} \begin{pmatrix} q_s \\ g \end{pmatrix} = \begin{pmatrix} P_{qq} & P_{qg} \\ P_{gq} & P_{gg} \end{pmatrix} \otimes \begin{pmatrix} q_s \\ g \end{pmatrix}, \quad (2.9)$$

with P_{qq} given by

$$P_{qq} = P_{qq}^v + P_{q\bar{q}}^v + n_f (P_{qq}^s + P_{q\bar{q}}^s). \quad (2.10)$$

³In this section we use the number densities $f(x, \mu^2)$. In QCDNUM itself, however, we use $xf(x, \mu^2)$.

⁴In the DIS scheme f is defined such that the LO (quark-parton model) expression for the F_2 structure function is preserved at NLO. But this is true only for F_2 and not for F_L and xF_3 .

Likewise, we find that the non-singlet combinations

$$q_{ij}^{\pm} = (q_i \pm \bar{q}_i) - (q_j \pm \bar{q}_j) \quad \text{and} \quad q_v = \sum_{i=1}^{n_f} (q_i - \bar{q}_i) \quad (2.11)$$

evolve independently from the gluon and from each other according to

$$\frac{\partial q_{ij}^{\pm}}{\partial \ln \mu^2} = P_{\pm} \otimes q_{ij}^{\pm} \quad \text{and} \quad \frac{\partial q_v}{\partial \ln \mu^2} = P_v \otimes q_v, \quad (2.12)$$

with splitting functions defined by

$$P_{\pm} = P_{qq}^v \pm P_{q\bar{q}}^v \quad \text{and} \quad P_v = P_{qq}^v - P_{q\bar{q}}^v + n_f(P_{qq}^s - P_{q\bar{q}}^s). \quad (2.13)$$

The evolution of the q_{ij}^{\pm} is linear in the densities, so that any linear combination of the q_{ij}^+ or q_{ij}^- also evolves according to (2.12).

The splitting functions can be expanded in a perturbative series in α_s which presently is known up to NNLO. For the four splitting functions P_{ij} in (2.9) we may write

$$P_{ij}(x, \mu^2) = a_s(\mu^2) P_{ij}^{(0)}(x) + a_s^2(\mu^2) P_{ij}^{(1)}(x) + a_s^3(\mu^2) P_{ij}^{(2)}(x) + O(a_s^4) \quad (2.14)$$

where we have set, as in the previous section, $a_s = \alpha_s/2\pi$. Note the separation in the variables x and μ^2 on the right-hand side of (2.14). We drop again the arguments x and μ^2 and write the expansion of the non-singlet splitting functions as

$$\begin{aligned} P_{\pm} &= a_s P_{qq}^{(0)} + a_s^2 P_{\pm}^{(1)} + a_s^3 P_{\pm}^{(2)} + O(a_s^4) \\ P_v &= a_s P_{qq}^{(0)} + a_s^2 P_v^{(1)} + a_s^3 P_v^{(2)} + O(a_s^4). \end{aligned} \quad (2.15)$$

Truncating the right-hand side to the appropriate order in a_s , it is seen that at LO the three types of non-singlet obey the same evolution equations. At NLO, q_{ij}^+ and q_v evolve in the same way but different from q_{ij}^- . At NNLO, all three non-singlets evolve differently.

It is evident from (2.7), (2.10) and (2.13) that several splitting functions depend on the number of active flavours n_f . This number is set to 3 below $\mu_F^2 = \mu_c^2$ and changed to $n_f = (4, 5, 6)$ at and above the thresholds $\mu_F^2 = \mu_{c,b,t}^2$. In case $\mu_F^2 \neq \mu_R^2$, QCDNUM adjusts the μ_R^2 thresholds such that n_f changes in both the splitting and the beta functions when crossing a threshold; see also Section 2.5.

The LO splitting functions are given in Appendix A. Those at NLO can be found in [12] (non-singlet) and [13] (singlet).⁵ The NNLO splitting functions and their parametrisations are given in [2] (non-singlet) and [3] (singlet). The DGLAP equations also apply to polarised parton densities and to fragmentation functions (time-like evolution), each with their own set of evolution kernels. For the polarised splitting functions up to NLO we refer to [14], and references therein. The time-like evolution of fragmentation functions at LO is described in [15], see also Appendix A. The NLO time-like splitting functions can be found in [12] and [13].

⁵Two well-known misprints in [13] are: (i) the lower integration limit in the definition of $S_2(x)$ must read $x/(1+x)$; (ii) in the expression for $\hat{P}_{FF}^{(1,T)}$ the term $(10 - 18x - \frac{16}{3}x^2)$ must read $(-10 - 18x - \frac{16}{3}x^2)$.

2.3 Renormalisation Scale Dependence

In the previous section, we have assumed that the factorisation and renormalisation scales are equal. For $\mu_F^2 \neq \mu_R^2$ we expand a_s in a Taylor series on a logarithmic scale around μ_R^2

$$a_s(\mu_F^2) = a_s(\mu_R^2) + a'_s(\mu_R^2)L_R + \frac{1}{2} a''_s(\mu_R^2)L_R^2 + \dots \quad (2.16)$$

with $L_R = \ln(\mu_F^2/\mu_R^2)$. Using (2.1) to calculate the derivatives in (2.16), we obtain

$$\begin{aligned} a_s(\mu_F^2) &= a_s(\mu_R^2) - \beta_0 L_R a_s^2(\mu_R^2) - (\beta_1 L_R - \beta_0^2 L_R^2) a_s^3(\mu_R^2) + O(a_s^4) \\ a_s^2(\mu_F^2) &= a_s^2(\mu_R^2) - 2\beta_0 L_R a_s^3(\mu_R^2) + O(a_s^4) \\ a_s^3(\mu_F^2) &= a_s^3(\mu_R^2) + O(a_s^4). \end{aligned} \quad (2.17)$$

To calculate the renormalisation scale dependence of the evolved parton densities, the powers of a_s in the splitting function expansions (2.14) and (2.15) are replaced by the expressions on the right-hand side of (2.17), with the understanding that these are truncated to order a_s when we evolve at LO, to order a_s^2 when we evolve at NLO, and to order a_s^3 when we evolve at NNLO.

2.4 Decomposition into Singlet and Non-singlets

In this section we describe the transformations between a flavour basis and a singlet/non-singlet basis, as is implemented in QCDNUM. For this purpose we write an arbitrary linear combination of quark and anti-quark densities as

$$|p\rangle = \sum_{i=1}^{n_f} (\alpha_i |q_i\rangle + \beta_i |\bar{q}_i\rangle), \quad (2.18)$$

where the index i runs over the number of active flavours. To make a clear distinction between a coefficient and a pdf, we introduce here the ket notation $|f\rangle$ for $f(x, \mu^2)$.

Because a linear combination of non-singlets is again a non-singlet, it follows directly from the definition (2.11) that the coefficients of any non-singlet satisfy the constraint

$$\sum_{i=1}^{n_f} (\alpha_i + \beta_i) = 0, \quad (2.19)$$

that is, a non-singlet is—by definition—orthogonal to the singlet in flavour space.

It is convenient to define $|q_i^\pm\rangle = |q_i\rangle \pm |\bar{q}_i\rangle$ and write the linear combination (2.18) as

$$|p\rangle = \sum_{i=1}^{n_f} (b_i^+ |q_i^+\rangle + b_i^- |q_i^-\rangle). \quad (2.20)$$

The coefficients b_i^\pm , α_i and β_i are related by

$$b_i^\pm = \frac{\alpha_i \pm \beta_i}{2}, \quad \alpha_i = b_i^+ + b_i^-, \quad \beta_i = b_i^+ - b_i^-. \quad (2.21)$$

We define a basis of singlet, valence, and $2(n_f - 1)$ additional non-singlets by

$$|e_1^+\rangle = |q_s\rangle, \quad |e_1^-\rangle = |q_v\rangle, \quad |e_i^\pm\rangle = \sum_{j=1}^{i-1} |q_j^\pm\rangle - (i-1) |q_i^\pm\rangle \quad \text{for } 2 \leq i \leq n_f. \quad (2.22)$$

In matrix notation, this transformation can be written as

$$|\mathbf{e}^\pm\rangle = \mathbf{U}|\mathbf{q}^\pm\rangle, \quad (2.23)$$

where \mathbf{U} is the $n_f \times n_f$ sub-matrix of the 6×6 transformation matrix

$$\mathcal{U} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & -2 & 0 & 0 & 0 \\ 1 & 1 & 1 & -3 & 0 & 0 \\ 1 & 1 & 1 & 1 & -4 & 0 \\ 1 & 1 & 1 & 1 & 1 & -5 \end{pmatrix}. \quad (2.24)$$

It is seen that the second to sixth row of (2.24) are orthogonal to the first row (singlet), so that they indeed represent non-singlets as defined by (2.19). In fact, all rows of \mathbf{U} are orthogonal to each other, so that scaling by the row-wise norm yields a rotation matrix, which has the transpose as its inverse. By scaling back this inverse we obtain

$$\mathbf{U}^{-1} = \mathbf{U}^T \mathbf{S}^2, \quad (2.25)$$

where \mathbf{U}^T is the transpose of \mathbf{U} and \mathbf{S}^2 is the square of the diagonal scaling matrix:

$$S_{ij}^2 = \delta_{ij} \left(\sum_{k=1}^{n_f} U_{ik}^2 \right)^{-1} = \begin{cases} \delta_{ij}/n_f & \text{for } i = 1 \\ \delta_{ij}/i(i-1) & \text{for } i > 1. \end{cases} \quad (2.26)$$

Using (2.25) and (2.26) to invert any $n_f \times n_f$ sub-matrix of (2.24), it is straight forward to show by explicit calculation that

$$U_{ij}^{-1} = \begin{cases} 1/n_f & \text{for } j = 1 \\ -1/j & \text{for } j = i \neq 1 \\ 1/j(j-1) & \text{for } j > i \\ 0 & \text{otherwise.} \end{cases} \quad (2.27)$$

The inverse of the transformation (2.22) is thus given by

$$\begin{aligned} |q_1^\pm\rangle &= \frac{|e_1^\pm\rangle}{n_f} + \sum_{j=2}^{n_f} \frac{|e_j^\pm\rangle}{j(j-1)} \\ |q_i^\pm\rangle &= \frac{|e_1^\pm\rangle}{n_f} - \frac{|e_i^\pm\rangle}{i} + \sum_{j=i+1}^{n_f} \frac{|e_j^\pm\rangle}{j(j-1)} \quad i > 1. \end{aligned} \quad (2.28)$$

We can now write the linear combination $|p\rangle$ on the $|e^\pm\rangle$ basis as

$$|p\rangle = \sum_{i=1}^{n_f} (d_i^+ |e_i^+\rangle + d_i^- |e_i^-\rangle), \quad (2.29)$$

where the coefficients d_i^\pm are related to the b_i^\pm of (2.20) by

$$d_i^\pm = \sum_{j=1}^{n_f} b_j^\pm U_{ji}^{-1}, \quad b_i^\pm = \sum_{j=1}^{n_f} d_j^\pm U_{ji}. \quad (2.30)$$

Let the starting values of the DGLAP evolutions be given by the gluon density and $2n_f$ arbitrary quark densities, that is, by $2n_f + 1$ functions of x at some input scale μ_0^2 . We can arrange the input quark densities in a $2n_f$ -dimensional vector $|\mathbf{p}\rangle$. Likewise, we store the densities $|q_i^\pm\rangle$ in a vector $|\mathbf{q}\rangle$, the $|e_i^\pm\rangle$ in a vector $|\mathbf{e}\rangle$ and the b^\pm coefficients of each input density in the rows of a $2n_f \times 2n_f$ matrix \mathbf{B} . The flavour decomposition of the input densities can then be written as $|\mathbf{p}\rangle = \mathbf{B}|\mathbf{q}\rangle$ and the singlet/non-singlet decomposition as

$$|\mathbf{p}\rangle = \mathbf{B}\mathbf{T}^{-1}|\mathbf{e}\rangle \quad \text{with} \quad \mathbf{T} \equiv \begin{pmatrix} \mathbf{U} & \mathbf{0} \\ \mathbf{0} & \mathbf{U} \end{pmatrix}. \quad (2.31)$$

Provided that \mathbf{B}^{-1} exists (*i.e.* the input densities are linearly independent), the starting values of the singlet and non-singlet densities are calculated from the inverse relation

$$|\mathbf{e}\rangle = \mathbf{T}\mathbf{B}^{-1}|\mathbf{p}\rangle. \quad (2.32)$$

2.5 Flavour Number Schemes

QCDNUM supports two evolution schemes, known as the fixed flavour number scheme (FFNS) and the variable flavour number scheme (VFNS).

In the FFNS we assume that n_f quark flavours have zero mass, while those of the remaining flavours are taken to be infinitely large. In this way, only n_f flavours participate in the QCD dynamics so that in the FFNS the value of n_f is simply kept constant for all μ^2 , with $3 \leq n_f \leq 6$. In the FFNS, the input scale μ_0^2 can be chosen anywhere within the boundaries of the evolution grid, although one should be careful with backward evolution in QCDNUM; see Section 3.3.

In the VFNS, the number of flavours changes from n_f to $n_f + 1$ when the factorisation scale is equal to the pole mass of the heavy quarks $\mu_h^2 = m_h^2$, $h = (c, b, t)$. A heavy quark h is thus considered to be infinitely massive below μ_h^2 and mass-less above μ_h^2 . As a consequence, the heavy flavour distributions are zero below their respective thresholds and are dynamically generated by the QCD evolution equations at and above μ_h^2 . Such an abrupt turn-on at a fixed scale is of course unphysical but this poses no problem since the parton densities themselves are not observables. The VFNS or FFNS parton densities evolved with QCDNUM are, in fact, valid input to structure function and cross section calculations that include mass terms and obey the kinematics of heavy quark production [16, 17, 18]. Such calculations are not part of QCDNUM itself, but can be coded in add-on packages; see Section 6.

An important feature of VFNS evolution is that the input scale μ_0^2 cannot be above the lowest heavy flavour threshold μ_c^2 . This is because otherwise heavy flavour contributions must be included in the input parton densities which clearly is in conflict with the dynamic generation of heavy flavour by the QCD evolution equations.

Another feature of the VFNS is the existence of discontinuities at the flavour thresholds in α_s and the parton densities; we will now turn to the calculation of these discontinuities. Because the beta functions (2.2) depend on n_f , it follows that the slope of the α_s evolution is discontinuous when crossing a threshold in the VFNS. Beyond LO there are not only discontinuities in the slope but also in α_s itself. In N^ℓLO, the value of $\alpha_s^{(n_f+1)}$ is, at a flavour threshold, related to $\alpha_s^{(n_f)}$ by [11, 19]

$$a_s^{(n_f+1)}(\kappa\mu_h^2) = a_s^{(n_f)}(\kappa\mu_h^2) + \sum_{n=1}^{\ell} \left\{ \left[a_s^{(n_f)}(\kappa\mu_h^2) \right]^{n+1} \sum_{j=0}^n C_{n,j} \ln^j \kappa \right\} \quad \ell = 1, 2. \quad (2.33)$$

Here μ_h^2 is the threshold defined on the factorisation scale and κ is the ratio μ_R^2/μ_F^2 at μ_h^2 . For $a_s = \alpha_s/4\pi$, the coefficients C in (2.33) read

$$C_{1,0} = 0, \quad C_{1,1} = \frac{2}{3}, \quad C_{2,0} = \frac{14}{3}, \quad C_{2,1} = \frac{38}{3}, \quad C_{2,2} = \frac{4}{9}.$$

Note that there is always a discontinuity in α_s at NNLO. At NLO, a discontinuity only occurs when $\kappa \neq 1$, that is, when the renormalisation and factorisation scales are different. In case of upward evolution, $\alpha_s^{(n_f+1)}$ is computed directly from (2.33) while for downward evolution, $\alpha_s^{(n_f-1)}$ is evaluated by numerically solving the equation

$$a_s^{(n_f)} - a_s^{(n_f-1)} - \Delta a_s \left(a_s^{(n_f-1)} \right) = 0,$$

where the function $\Delta a_s(a_s)$ is given by the second term on the right-hand side of (2.33).

In the VFNS at NNLO, not only α_s but also the parton densities have discontinuities at the flavour thresholds [20]:

$$\begin{aligned} g(x, \mu_h^2, n_f + 1) &= g(x, \mu_h^2, n_f) + \Delta g(x, \mu_h^2, n_f) \\ q_i^\pm(x, \mu_h^2, n_f + 1) &= q_i^\pm(x, \mu_h^2, n_f) + \Delta q_i^\pm(x, \mu_h^2, n_f) \quad i = 1, \dots, n_f \\ h^+(x, \mu_h^2, n_f + 1) &= \Delta h^+(x, \mu_h^2, n_f) \\ h^-(x, \mu_h^2, n_f + 1) &= \Delta h^-(x, \mu_h^2, n_f) = 0, \end{aligned} \quad (2.34)$$

where $h = (c, b, t)$ for $n_f = (3, 4, 5)$. Note that a heavy quark h becomes a light quark q_i above the threshold μ_h^2 .

In QCDNUM, the flavour thresholds on the renormalisation scale are adjusted such that n_f changes by one unit in both the beta functions and the splitting functions when crossing a threshold. With this choice, the parton densities are continuous at LO and NLO while at NNLO the calculation of the discontinuities is considerably simplified (all terms proportional to powers of $\ln(m^2/\mu^2)$ in ref. [20] vanish). So we may write

$$\begin{aligned} \Delta g(x, \mu_h^2, n_f) &= a_s^2 \{ [A_{gq} \otimes q_s](x, \mu_h^2, n_f) + [A_{gg} \otimes g](x, \mu_h^2, n_f) \} \\ \Delta q_i^\pm(x, \mu_h^2, n_f) &= a_s^2 [A_{qq} \otimes q_i^\pm](x, \mu_h^2, n_f) \\ \Delta h^+(x, \mu_h^2, n_f) &= a_s^2 \{ [A_{hq} \otimes q_s](x, \mu_h^2, n_f) + [A_{hg} \otimes g](x, \mu_h^2, n_f) \}. \end{aligned} \quad (2.35)$$

Here a_s stands for $a_s^{(n_f+1)}(\kappa\mu_h^2)$ as defined by (2.33). The convolution kernels A_{ij} can be found in Appendix B of [20].⁶

⁶In the notation of [20], $A_{gq} = A_{gq,H}^{S,(2)}$ (eq. B.5), $A_{gg} = A_{gg,H}^{S,(2)}$ (B.7), $A_{qq} = A_{qq,H}^{NS,(2)}$ (B.4), $A_{hq} = \tilde{A}_{Hq}^{PS,(2)}$ (B.1) and $A_{hg} = \tilde{A}_{Hg}^{S,(2)}$ (B.3). For the latter we use a parametrisation provided by A. Vogt.

The discontinuities in the basis vectors $|e_i^\pm\rangle$ are calculated from

$$e_i^\pm(x, \mu_h^2, n_f + 1) = e_i^\pm(x, \mu_h^2, n_f) + \Delta e_i^\pm(x, \mu_h^2, n_f) + \lambda_i(n_f) \Delta h^\pm(x, \mu_h^2, n_f), \quad (2.36)$$

where the light component Δe_i^\pm is given by (2.35), with q_i^\pm replaced by e_i^\pm . With the definition (2.22) of the basis functions, the values of the coefficients $\lambda_i(n_f)$ are

n_f	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6
3	1	0	0	-3		
4	1	0	0	0	-4	
5	1	0	0	0	0	-5

(2.37)

When the densities are evolved upward in μ^2 , it is straight forward to calculate with (2.34) and (2.35) the parton densities at $n_f + 1$ from those at n_f . However, QCDNUM is capable to invert the relation between n_f and $n_f + 1$ so that it can also calculate the discontinuities in case of downward evolution. For this it is convenient to write the calculation of the singlet and gluon discontinuities in matrix form, similar to (2.9)

$$\begin{pmatrix} q_s \\ g \end{pmatrix}^{(n_f+1)} = \begin{pmatrix} q_s \\ g \end{pmatrix}^{(n_f)} + a_s^2 \begin{pmatrix} A_{qq} + A_{hq} & A_{hg} \\ A_{gq} & A_{gg} \end{pmatrix} \otimes \begin{pmatrix} q_s \\ g \end{pmatrix}^{(n_f)}. \quad (2.38)$$

In Section 3.3 we will show how (2.38) is turned into an invertible matrix equation.

Note that the heavy quark non-singlets do not obey the DGLAP evolution equations over the full range in μ^2 , because the heavy flavours are simply set to zero below their thresholds, instead of being evolved. The evolution of the set $|e_i^\pm\rangle$ thus proceeds in the VFNS as follows: The singlet/valence densities $|e_1^\pm\rangle$ and the light non-singlets $|e_{2,3}^\pm\rangle$ are evolved both upward and downward starting from some scale $\mu_0^2 < \mu_c^2$. The heavy non-singlets $|e_{4,5,6}^\pm\rangle$ are dynamically generated from the DGLAP equations by upward evolution from the thresholds $\mu_{c,b,t}^2$. At and below the thresholds, $|e_{4,5,6}^\pm\rangle$ is set equal to the singlet and $|e_{4,5,6}^-\rangle$ to the valence. This is equivalent to setting the heavy quark and anti-quark distributions to zero, except that at NNLO the heavy flavours do not evolve from zero but from the non-zero discontinuity given in (2.34). This is illustrated in Figure 2 where we plot the charm and bottom starting distributions, normalised to the singlet distribution. It is seen that the bottom discontinuity is less than 3% of the singlet over the whole range in x , while for charm it is much larger, exceeding 10% at low x . Note that the starting distributions are negative below $x \approx 10^{-2}$.

3 Numerical Method

The DGLAP evolution equations are in QCDNUM numerically solved on a discrete $n \times m$ grid in x and μ^2 . In such an approach the convolution integrals can be evaluated as weighted sums with weights calculated once and for all at program initialisation. Because of the convolutions, the total number of operations to solve a DGLAP equation is quadratic in n and linear in m . The accuracy of the solution depends, for a given grid, on the interpolation scheme chosen (linear or quadratic).

The advantage of this ‘ x -space’ approach, compared to ‘ N -space’ [19], is its conceptual simplicity and the fact that one is completely free to chose the functional form of the

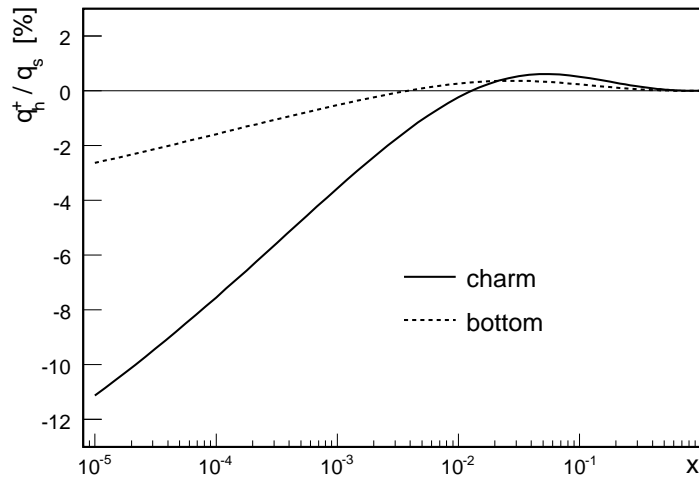


Figure 2: The NNLO starting densities $q_h^+(x, \mu_h^2)$, normalised to the singlet density $q_s(x, \mu_h^2)$, for charm (full curve) and bottom (dotted curve).

input distribution since it is fed into the evolution as a discrete vector of input values. A disadvantage is that accuracy and speed depend on the choice of grid and that each evolution will yield no less than $n \times m$ parton density values (typically 10^4) whether you want them or not.

The numerical method used in QCDNUM is based on polynomial spline interpolation of the parton densities on an equidistant logarithmic grid in x and a (not necessarily equidistant) logarithmic grid in μ^2 . The order of the x -interpolation can in be set to $k = 2$ (linear) or 3 (quadratic). The interpolation in μ^2 is always quadratic. With such an interpolation scheme, the DGLAP evolution equations transform into a triangular set of linear equations in the interpolation coefficients. This leads to a very fast evolution of these coefficients from some input scale μ_0^2 to any other scale μ_i^2 on the grid. In the following sections we will describe the spline interpolation, the calculation of convolution integrals and the QCD evolution algorithm. Note that several features of the QCDNUM17 numerical method have been previously proposed in, for example, [21, 22].

3.1 Polynomial Spline Interpolation

To interpolate a function $h(y)$,⁷ we sample this function on an $(n + 1)$ -point grid

$$y_0 < y_1 < \dots < y_{n-1} < y_n$$

and parametrise it in each interval by a piecewise polynomial of order k . Such a piecewise polynomial is turned into a spline by imposing one or more continuity relations at each of the grid points. Usually—but not always—continuity is imposed at the internal grid points on the function itself and on all but the highest derivative, which is allowed to be

⁷In QCDNUM, $h(y)$ represents a parton *momentum* density in the scaling variable $y = -\ln x$. However, for this section the identification of h with a parton density is not so relevant.

discontinuous. Without further constraints at the end points, the spline has $k + n - 1$ free parameters. Increasing the order k of the interpolation thus costs only *one* and not n extra parameters as is the case for unconstrained piecewise polynomials.

It is convenient to write a spline function as a linear combination of so-called B-splines

$$h(y) = \sum_i A_i Y_i(y). \quad (3.1)$$

The basis Y_i of B-splines depends on the order k , on the distribution of the grid points along the y axis (equidistant in QCDNUM) and on the number of continuity relations we wish to impose at the internal grid points and at the two end points. For how to construct a B-spline basis and for more details on splines in general we refer to [23].

In Figure 3 are shown the B-splines for linear ($k = 2$) quadratic ($k = 3$) and cubic

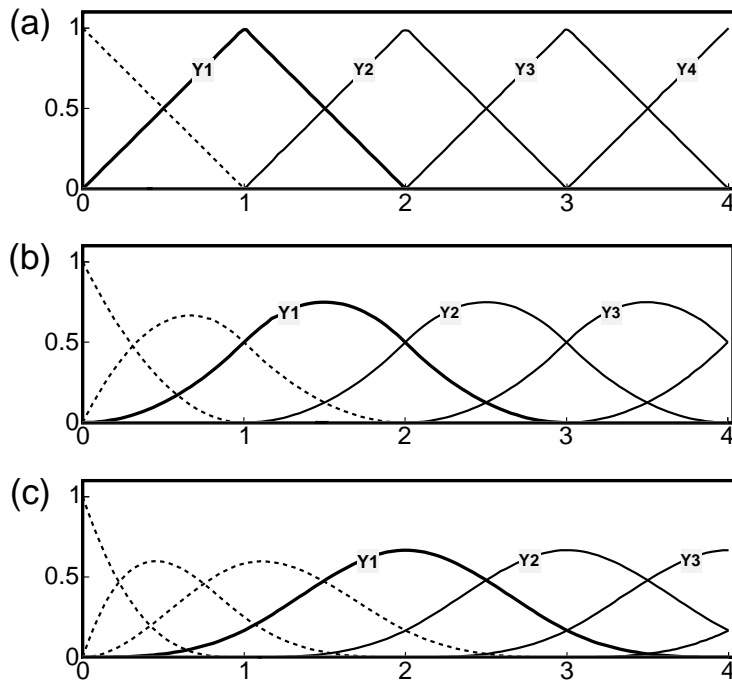


Figure 3: B-spline bases generated on an equidistant grid. (a) Linear B-splines ($k = 2$). Removing the dashed spline enforces the boundary condition $h(y_0) = 0$; (b) Quadratic B-splines ($k = 3$). Removing the first two dashed splines enforces the boundary condition $h(y_0) = h'(y_0) = 0$; (c) Cubic B-splines ($k = 4$). Removing the first three dashed splines enforces the boundary condition $h(y_0) = h'(y_0) = h''(0) = 0$. Spline interpolation on such a basis is numerically unstable.

($k = 4$) interpolation on an equidistant grid. In case $h(y_0) = h(0) = 0$ —which is always true for parton densities—we may remove the first B-spline in the plots of Figure 3. Removing the second B-spline in Figure 3b gives quadratic interpolation with an additional boundary condition $h'(y_0) = 0$.⁸ With these boundary conditions—and because

⁸A parton density parametrisation should thus behave like $h(y \rightarrow 0) \propto y^\lambda$ with $\lambda > 1$ because otherwise the condition $h'(0) = 0$ is violated and the spline might oscillate. All known pdf parametrisations fulfil this requirement but when the parameters are under control of a fitting program one should take precautions that λ will always stay above unity.

the grid is equidistant—the remaining B-splines possess translation invariance, that is, the basis can be generated by successively shifting the first spline one interval to the right (full curves in Figure 3a,b). Translation invariance greatly simplifies the evolution algorithm, as we will see later.

It is therefore tempting to extend the scheme to cubic interpolation by removing the first three B-splines in Figure 3c. This would yield a translation invariant basis with the boundary conditions $h(y_0) = h'(y_0) = h''(y_0) = 0$. However, it turns out that such a cubic spline interpolation tends to be numerically unstable. The cure is to drop the constraint $h''(y_0) = 0$ and impose a constraint on $h'(y_n)$ at the other end of the grid. But this does not fit in the evolution algorithm as it now stands so that we have abandoned cubic and higher order splines in QCDNUM.

If we number the B-splines $1, 2, \dots, n$ from left to right as indicated in Figure 3 it is seen that for both $k = 2$ and 3 the following relation holds (translation invariance):

$$Y_i(y) = Y_1(y - y_{i-1}). \quad (3.2)$$

Furthermore, for linear interpolation ($k = 2$) we have $Y_i(y_i) = 1$ so that

$$\begin{aligned} h(y_0) &= 0 \\ h(y_i) &= A_i Y_i(y_i) = A_i \quad 1 \leq i \leq n. \end{aligned} \quad (3.3)$$

Likewise, for quadratic interpolation ($k = 3$) we have $Y_{i-1}(y_i) = Y_i(y_i) = 1/2$ so that

$$\begin{aligned} h(y_0) &= 0 \\ h(y_1) &= A_1 Y_1(y_1) = A_1/2 \\ h(y_i) &= A_{i-1} Y_{i-1}(y_i) + A_i Y_i(y_i) = (A_{i-1} + A_i)/2 \quad 2 \leq i \leq n. \end{aligned} \quad (3.4)$$

We denote $h(y_i)$ by h_i , the column vector of function values by $\mathbf{h} = (h_1, \dots, h_n)^T$, the corresponding vector of spline coefficients by \mathbf{a} and write (3.3) and (3.4) as

$$\mathbf{h} = \mathbf{S} \mathbf{a} \quad (3.5)$$

where \mathbf{S} is the identity matrix in case of linear interpolation and a lower diagonal band matrix for the quadratic spline. On a 5-point equidistant grid y_0, \dots, y_4 , for instance, we have in case of quadratic interpolation the vector $\mathbf{h} = (h_1, \dots, h_4)^T$ and the matrix

$$\mathbf{S} = \frac{1}{2} \begin{pmatrix} 1 & & & & \\ 1 & 1 & & & \\ & 1 & 1 & & \\ & & 1 & 1 & \\ & & & 1 & 1 \end{pmatrix} \quad \text{with inverse} \quad \mathbf{S}^{-1} = 2 \begin{pmatrix} 1 & & & & \\ -1 & 1 & & & \\ 1 & -1 & 1 & & \\ -1 & 1 & -1 & 1 & \end{pmatrix}. \quad (3.6)$$

Note that \mathbf{S} is sparse but \mathbf{S}^{-1} is not. Thus, when a parton distribution \mathbf{h}_0 is given at some input scale μ_0^2 , the corresponding vector \mathbf{a}_0 of spline coefficients is found by solving (3.5).⁹ This vector is then evolved to other values of μ^2 using the DGLAP evolution equations as is described in the next two sections.

⁹Obtaining \mathbf{a} from solving (3.5) by forward substitution (Appendix B) costs $O(2n)$ operations. This is cheaper than the alternative of calculating $\mathbf{a} = \mathbf{S}^{-1} \mathbf{h}$ which costs $O(n^2/2)$ operations.

3.2 Convolution Integrals

The Mellin convolution (2.5) calculated in QCDNUM is not that of a number density f and some kernel g but, instead, that of a momentum density $p = xf$ and a kernel $q = xg$. These convolutions differ by a factor x :

$$[p \otimes q](x) = x[f \otimes g](x). \quad (3.7)$$

This is also true for multiple convolution: for $p = xf$, $q = xg$ and $r = xh$ we have

$$[p \otimes q \otimes r](x) = x[f \otimes g \otimes h](x). \quad (3.8)$$

A change of variable $y = -\ln x$ turns a Mellin convolution into a Fourier convolution:

$$[f \otimes g](x) = [u \otimes v](y) = \int_0^y dz u(z) v(y-z) = \int_0^y dz u(y-z) v(z), \quad (3.9)$$

where the functions u and v are defined by $u(y) = f(e^{-y})$ and $v(y) = g(e^{-y})$.

In the following we will denote by $h(y, t)$ a parton *momentum* density in the logarithmic scaling variables $y = -\ln x$ and $t = \ln \mu^2$. In terms of h , the DGLAP non-singlet evolution equation (2.12) is written as

$$\frac{\partial h(y, t)}{\partial t} = \int_0^y dz Q(z, t) h(y-z, t) = \int_0^y dz Q(y-z, t) h(z, t) \quad (3.10)$$

with a kernel $Q(y, t) = e^{-y} P(e^{-y}, t)$. Here $P(x, t)$ is a non-singlet splitting function, as given in Section 2.2. To solve (3.10) we first have to evaluate the Fourier convolution

$$I(y, t) \equiv \int_0^y dz Q(y-z, t) h(z, t). \quad (3.11)$$

Inserting (3.1) in (3.11) we find for the integrals at the grid points y_i (for clarity, we drop the argument t in the following)

$$I(y_i) = \sum_{j=1}^i A_j \int_0^{y_i} dz Q(y_i - z) Y_j(z) \equiv \sum_{j=1}^i W_{ij} A_j \quad (1 \leq i \leq n). \quad (3.12)$$

The summation is over the first i terms only, because B-splines with an index $j > i$ are zero in the integration domain $z \leq y_i$, see Figure 3.

Eq. (3.12) defines the weights W_{ij} which are calculated as follows. Because $Y_j(y) = 0$ for $y < y_{j-1}$ the weights can be written as

$$W_{ij} = \int_{y_{j-1}}^{y_i} dz Q(y_i - z) Y_j(z) = \int_0^{y_i - y_{j-1}} dz Q(y_i - y_{j-1} - z) Y_1(z) \quad (3.13)$$

where we have used (3.2) in the second identity. From the property of equidistant grids

$$y_i + y_j = y_{i+j}$$

it follows that W_{ij} depends only on the difference $i - j$ (Toeplitz matrix):

$$W_{ij} = w_{i-j+1} \quad \text{with} \quad w_\ell \equiv \int_0^{y_\ell} dz Q(y_\ell - z) Y_1(z) \quad (1 \leq \ell \leq n). \quad (3.14)$$

The integrand only contributes in the region $k\Delta$ where Y_1 is non-zero so that in practical calculations the upper integration limit y_ℓ is replaced by $\min(y_\ell, k\Delta)$, with Δ the grid spacing. We remark that the calculation of the weights w_ℓ is a bit more complicated than suggested by (3.14) because singularities in the splitting functions have to be taken into account; for the relevant formula's we refer to Appendix A.

The weights can thus be arranged in a lower-triangular Toeplitz matrix, as is illustrated by the 4×4 example below:

$$W_{ij} = \begin{pmatrix} w_1 & & & \\ w_2 & w_1 & & \\ w_3 & w_2 & w_1 & \\ w_4 & w_3 & w_2 & w_1 \end{pmatrix}. \quad (3.15)$$

This matrix is fully specified by the first column, taking n instead of $n(n+1)/2$ words of storage. This is not only advantageous in terms of memory usage but also in terms of computing speed since frequent calculations like summing the perturbative expansion

$$\mathbf{W}(t) = a_s(t) \{ \mathbf{W}^{(0)} + a_s(t) \mathbf{W}^{(1)} + \dots \} \quad (3.16)$$

takes only $O(n)$ operations instead of $O(n^2/2)$. We write the vector of convolution integrals as \mathbf{I} and express (3.12) in vector notation as

$$\mathbf{I} = \mathbf{W} \mathbf{a}. \quad (3.17)$$

Also multiple convolutions can be calculated as weighted sums. Let $f(x)$ be a number density and $K_{a,b}(x)$ be two convolution kernels. The vector of Mellin convolutions

$$I_i = x_i [f \otimes K_a \otimes K_b](x_i)$$

can be calculated from (3.17), using the weight table

$$\mathbf{W} = \mathbf{W}_a \mathbf{S}^{-1} \mathbf{W}_b. \quad (3.18)$$

Here \mathbf{W}_a and \mathbf{W}_b are the weight tables of K_a and K_b , respectively, and \mathbf{S} is the transformation matrix defined by (3.5).

Another interesting convolution is that of two number densities f_a and f_b

$$I_i = x_i [f_a \otimes f_b](x_i).$$

This ‘parton luminosity’ [24] (times x) is calculated from the Fourier convolution

$$I(y_i) = \int_0^{y_i} dz h_a(z) h_b(y_i - z). \quad (3.19)$$

Inserting the spline representation (3.1) gives an expression for the convolution integral as a weighted sum over the set of spline coefficients \mathbf{a} of h_a and \mathbf{b} of h_b ,

$$I(y_i) = \sum_{j=1}^i \sum_{k=1}^i A_j B_k W_{ijk} \quad \text{with} \quad W_{ijk} \equiv \int_0^{y_i} dz Y_j(z) Y_k(y_i - z).$$

To reduce the dimension of W_{ijk} , we use the translation invariance (3.2) and write

$$W_{ijk} = \int_0^{y_{i-j+1}} dz Y_1(z) Y_k(y_{i-j+1} - z).$$

Because B-splines with index $k > i-j+1$ do not have their support inside the integration domain, we obtain an upper limit $k \leq i-j+1$. Again using translation invariance yields

$$W_{ijk} = \int_0^{y_{i-j-k+2}} dz Y_1(z) Y_1(y_{i-j-k+2} - z).$$

We now have a compact expression for the convolution integral (3.19):

$$I(y_i) = \sum_{j=1}^i \sum_{k=1}^{i-j+1} A_j B_k w_{i-j-k+2} \quad \text{with} \quad w_\ell = \int_0^{y_\ell} dz Y_1(z) Y_1(y_\ell - z). \quad (3.20)$$

Because Y_1 has a limited support, it turns out that only the first 3 (5) terms of w_ℓ are non-zero in case of linear (quadratic) interpolation. The operation count to calculate a convolution of parton densities is thus not more than $O(5n)$, for quadratic splines.

3.3 DGLAP Evolution

We denote by the vector \mathbf{h}_0 a *non-singlet* quark density at the input scale $t_0 = \ln \mu_0^2$. The derivative of \mathbf{h}_0 with respect to the scaling variable t is given by the DGLAP evolution equation (3.10) which can be written in vector notation as, from (3.5) and (3.17)

$$\frac{d\mathbf{h}_0}{dt} = \frac{d\mathbf{S}\mathbf{a}_0}{dt} = \mathbf{W}_0 \mathbf{a}_0 \quad \text{or} \quad \frac{d\mathbf{a}_0}{dt} \equiv \mathbf{a}'_0 = \mathbf{S}^{-1} \mathbf{W}_0 \mathbf{a}_0. \quad (3.21)$$

Likewise we have at t_1

$$\mathbf{a}'_1 = \mathbf{S}^{-1} \mathbf{W}_1 \mathbf{a}_1. \quad (3.22)$$

We have indexed the weight matrices above by a subscript because they depend on t through multiplication by powers of a_s , see (3.16).

Assuming that $\mathbf{a}(t)$ is quadratic in t , we can relate \mathbf{a}_0 , \mathbf{a}_1 , \mathbf{a}'_0 and \mathbf{a}'_1 by

$$\mathbf{a}_1 = \mathbf{a}_0 + (\mathbf{a}'_0 + \mathbf{a}'_1) \Delta_1 \quad (3.23)$$

with $\Delta_1 = (t_1 - t_0)/2$. If $t_1 > t_0$, Δ_1 is positive and we perform forward evolution. If $t_1 < t_0$, Δ_1 is negative and we perform backward evolution.

Inserting (3.21) and (3.22) in (3.23) we obtain a relation between the known spline coefficients \mathbf{a}_0 and the unknown coefficients \mathbf{a}_1

$$(\mathbf{1} - \mathbf{S}^{-1} \mathbf{W}_1 \Delta_1) \mathbf{a}_1 = (\mathbf{1} + \mathbf{S}^{-1} \mathbf{W}_0 \Delta_1) \mathbf{a}_0. \quad (3.24)$$

Multiplying both sides from the left by $\mathbf{U}_1 \equiv \mathbf{S}/\Delta_1$ gives

$$(\mathbf{U}_1 - \mathbf{W}_1) \mathbf{a}_1 = (\mathbf{U}_1 + \mathbf{W}_0) \mathbf{a}_0. \quad (3.25)$$

Eq. (3.25) is more convenient than (3.24) because matrix multiplication $\mathbf{S}^{-1}\mathbf{W}$ is replaced by matrix addition.¹⁰ Note that \mathbf{U} is a lower diagonal band matrix so that $\mathbf{U} \pm \mathbf{W}$ is still lower triangular with, in fact, the Toeplitz structure (3.15) preserved. All this leads to a very simple and fast evolution algorithm, starting from \mathbf{a}_0 :

1. At t_0 , calculate \mathbf{a}_0 from (3.5), \mathbf{W}_0 from (3.16) and \mathbf{U}_1 as defined above. Then construct the vector $\mathbf{b}_1 \equiv (\mathbf{U}_1 + \mathbf{W}_0) \mathbf{a}_0$.
2. Subsequently, at t_1 ,
 - (a) Calculate \mathbf{W}_1 and the lower triangular matrix $\mathbf{V}_1 = \mathbf{U}_1 - \mathbf{W}_1$;
 - (b) Solve the equation $\mathbf{V}_1 \mathbf{a}_1 = \mathbf{b}_1$ by forward substitution, see Appendix B;
 - (c) Calculate \mathbf{U}_2 and $\mathbf{b}_2 = (\mathbf{U}_1 + \mathbf{U}_2) \mathbf{a}_1 - \mathbf{b}_1$ for the next evolution to t_2 .¹¹
3. Repeat step 2 at t_2 and so on.

With this algorithm each evolution step consists of a few vector manipulations which have an operation count $O(n)$ and solving one triangular matrix equation which has an operation count $O(n^2/2)$. The total operation count only very weakly depends on the order k of the interpolation chosen: quadratic interpolation is almost for free.

The algorithm can also be used for the coupled evolution of the singlet quark (\mathbf{a}_s) and gluon (\mathbf{a}_g) spline coefficients, provided we make the following replacements in the formalism:

$$\mathbf{a} \rightarrow \begin{pmatrix} \mathbf{a}_s \\ \mathbf{a}_g \end{pmatrix} \quad \mathbf{S} \rightarrow \begin{pmatrix} \mathbf{S} & \\ & \mathbf{S} \end{pmatrix} \quad \mathbf{W} \rightarrow \begin{pmatrix} \mathbf{W}_{qq} & \mathbf{W}_{qg} \\ \mathbf{W}_{gq} & \mathbf{W}_{gg} \end{pmatrix}.$$

In Appendix B is shown how the coupled triangular equations are solved by extending the forward substitution algorithm. The operation count is $4 \times O(n^2/2)$ so that for m grid points in t we have in total $O(2n^2m)$ operations for the singlet-gluon evolution and $O(n^2m/2)$ operations for each non-singlet evolution.

Finally, let us express in vector notation the NNLO parton density discontinuities at the flavour thresholds. The relation between the singlet and gluon distributions at n_f and $n_f + 1$ as given by (2.38) can be written as

$$\begin{pmatrix} \mathbf{S} & \\ & \mathbf{S} \end{pmatrix} \begin{pmatrix} \mathbf{a}_s \\ \mathbf{a}_g \end{pmatrix}^{(n_f+1)} = \begin{pmatrix} \mathbf{S} + \mathbf{A}_{qq} + \mathbf{A}_{hq} & \mathbf{A}_{hg} \\ \mathbf{A}_{gq} & \mathbf{S} + \mathbf{A}_{gg} \end{pmatrix} \begin{pmatrix} \mathbf{a}_s \\ \mathbf{a}_g \end{pmatrix}^{(n_f)}. \quad (3.26)$$

It is easy to solve this linear equation for $\mathbf{a}^{(n_f+1)}$ when $\mathbf{a}^{(n_f)}$ is known (forward evolution) or for $\mathbf{a}^{(n_f)}$ when $\mathbf{a}^{(n_f+1)}$ is known (backward evolution). Likewise, we may write for the non-singlet discontinuities

$$\mathbf{S} \mathbf{a}_{ns}^{(n_f+1)} = (\mathbf{S} + \mathbf{A}_{qq}) \mathbf{a}_{ns}^{(n_f)} + \lambda \left(\mathbf{A}_{hq} \mathbf{a}_s^{(n_f)} + \mathbf{A}_{hg} \mathbf{a}_g^{(n_f)} \right), \quad (3.27)$$

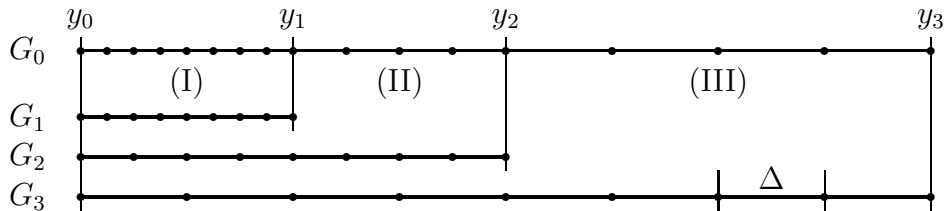
¹⁰In fact, adding a matrix with band structure (3.6) to a lower triangular matrix with structure (3.15) takes only *two* additions irrespective of the dimension of the matrices.

¹¹Using (3.25) it is a simple exercise to establish this relation between \mathbf{b} , \mathbf{U} and \mathbf{a} . Note that \mathbf{b} in step (2c) is calculated much faster than \mathbf{b} in step (1).

where λ is defined by (2.36). Also this equation can easily be inverted.

It can be seen from (3.5) and (3.23) that $h(y, t)$ is, by construction, a spline in both the variables y and t . However, it turns out that it is technically more convenient to represent the pdfs by their *values* on the grid, instead of by their spline coefficients. Polynomial interpolation of order k in y and quadratic in t is then done locally on a $k \times 3$ mesh around the interpolation point. The NNLO discontinuities are preserved by storing, at the flavour thresholds, the pdf values for both $n_f - 1$ and n_f , and by prohibiting the interpolation mesh to cross a flavour threshold. Note, however, that the interpolation routine yields a single-valued function of t , so that one has to calculate $h(y, t_{c,b,t} - \epsilon)$ to view the discontinuity.¹²

In QCDNUM it is possible to evolve on multiple equidistant y -grids which allow for a finer binning at low y (large x) where the parton densities are rapidly varying. This is illustrated below by a grid G_0 which is built-up from three equidistant sub-grids G_1 , G_2 and G_3 with spacing $\Delta/4$, $\Delta/2$ and Δ , respectively.



On such a multiple grid, the parton densities are first evolved on the grid G_1 and the results are copied to the region (I) of G_0 . The evolution is then repeated on the grids G_2 and G_3 followed by a copy to the regions (II) and (III) of G_0 , respectively. We refer to Section 4.3 for spectacular gains in accuracy that can be achieved by employing these multiple grids.

As remarked above, the evolution algorithm can—at least in principle—handle both forward and backward evolution in μ^2 simply by changing the sign of Δ in (3.23). This works very well for linear spline interpolation but it turns out that backward evolution of quadratic splines can sometimes lead to severe oscillations. This is illustrated in Figure 4 where is shown a non-singlet quark density evolved downward from $\mu_0^2 = 5$ to $\mu^2 = 2$ GeV² in the quadratic interpolation scheme (dotted curve). In QCDNUM this numerical instability is handled as follows: (i) evolve downward from μ_0^2 to μ^2 in the linear interpolation scheme (which is stable); (ii) then take μ^2 as the starting scale and evolve *upward* to μ_0^2 in the quadratic interpolation scheme (also stable); (iii) calculate the difference Δf between the newly evolved pdf and the original one at μ_0^2 ; (iv) subtract Δf from the starting value at μ_0^2 used in (i) and repeat the procedure.

The full curve in the top plot of Figure 4 shows the result of downward evolution in the linear interpolation scheme, that is, without iterations. Oscillations are absent but the evolution is not very accurate as is evident from the difference between the dotted and full curve at large x . One iteration already much improves the precision as can be seen from the good match at large x between the two curves in the bottom plot. It turns out that one iteration (QCDNUM default), perhaps two, are sufficient while more iterations

¹²Do not take ϵ too small because QCDNUM may snap to the threshold, see Section 5.2.

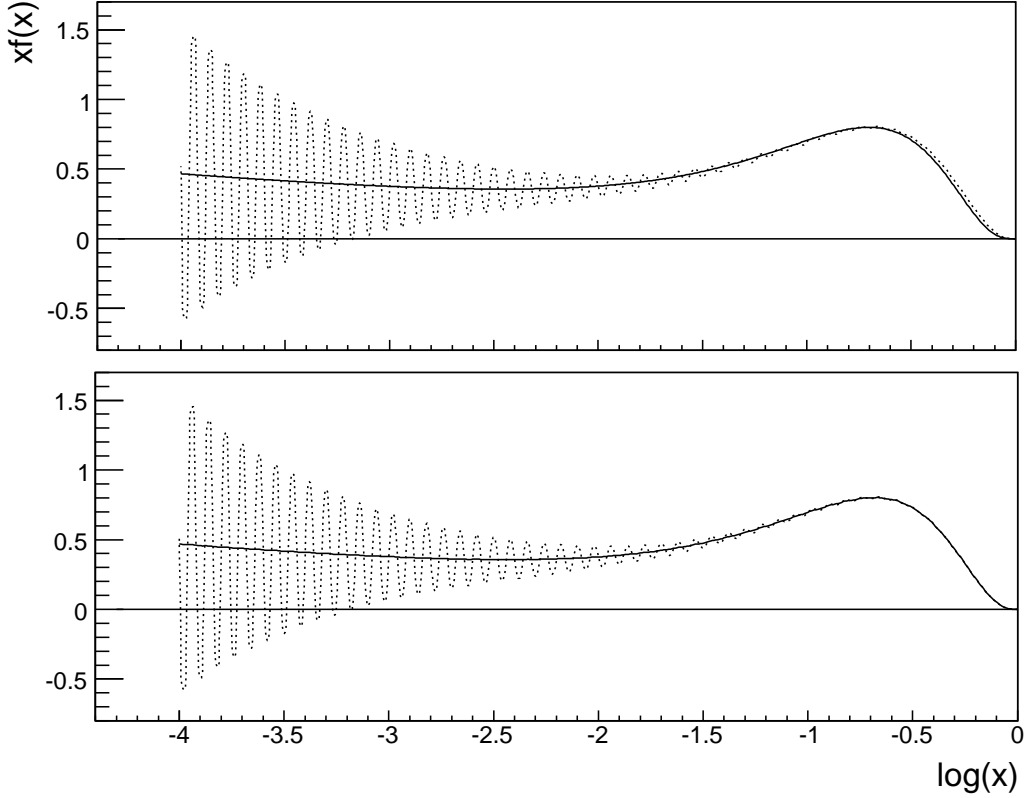


Figure 4: A non-singlet parton density $xf(x)$ versus $\log(x)$ evolved downward from $\mu_0^2 = 5$ to $\mu^2 = 2 \text{ GeV}^2$ in the quadratic interpolation scheme showing large oscillations (dotted curve). The full curve in the top plot shows the result of downward evolution in the linear interpolation scheme. The full curve in the bottom plot shows an improved result obtained by iteration, as described in the text.

tend to spoil the convergence. Clearly best is to limit the range of downward evolution by keeping μ_0^2 low or, if possible, to set it at the lowest grid point to avoid downward evolution altogether.

QCDNUM checks for quadratic spline oscillation as follows. We denote the values of the quadratic B-spline at $(\frac{1}{2}\Delta, \Delta, \frac{3}{2}\Delta)$ by $(b_1, b_2, b_3) = (\frac{1}{8}, \frac{1}{2}, \frac{3}{4})$. It is easy to show that quadratic interpolation mid-between the grid points is given by $\mathbf{u} = \mathbf{D}\mathbf{a}$, where \mathbf{D} is a lower diagonal Toeplitz band matrix, of bandwidth 3, which is characterised by the vector (b_1, b_3, b_1) . Likewise, the linear interpolation of the spline at the mid-points is calculated from $\mathbf{v} = \mathbf{E}\mathbf{a}$, where \mathbf{E} is the lower diagonal Toeplitz band matrix $(\frac{1}{2}b_2, b_2, \frac{1}{2}b_2)$. The maximum deviation $\epsilon = \|\mathbf{u} - \mathbf{v}\| = \|(\mathbf{D} - \mathbf{E})\mathbf{a}\|$ should be small; for pdfs sampled on a reasonably dense grid, $\epsilon \approx 0.1$ or less. For each pdf evolution, ϵ is computed at the input scale, and at the lower and upper end of the μ^2 grid. An error condition is raised when it exceeds a given limit, indicating that the spline oscillates, or that the x -grid is not dense enough.

4 The QCDNUM Program

4.1 Source Code

The QCDNUM source code can be downloaded from the web site

<http://www.nikhef.nl/user/h24/qcdnum>

Unpacking the tar file produces a directory `qcdnum-xx-yy` with `xx-yy` the version number. Sub-directories contain the source code, example jobs, write-up and a simple script to make a QCDNUM library, see the `README` file. The code comes with a utility package MBUTIL (including write-up) which is a collection of general-purpose routines (some developed privately, some taken from CERNLIB and some taken from public source code repositories like NETLIB). Because QCDNUM uses several of these routines, MBUTIL must also be compiled and linked to your application program. Apart from this, QCDNUM is completely stand-alone. To calculate structure functions, the ZMSTF and HQSTF add-on packages are provided, see Sections C.3 and D.1.

Before compiling QCDNUM you may want to set several parameters which control the size of internal arrays. These parameters can be found in the include file `qcdnum.inc`:

`mxg0` Maximum number of multiple x -grids [5].
`mxx0` Maximum number of points in the x -grid [300].
`mqq0` Maximum number of points in the μ^2 -grid [150].
`mpt0` Maximum number of interpolations calculated in a single call [5000].
`miw0` Maximum number of information words in a weight store [20].
`mbf0` Maximum number of fast convolution scratch buffers [20].
`nwf0` Size of the QCDNUM dynamic store in words [400000].

The first 6 parameters are simply dimensions of book-keeping arrays which you may want to adjust to your needs. More important is the parameter `nwf0` that defines the size of an internal store that contains the weight tables and the tables of parton densities. How many words are needed depends on the size of the tables which, in turn, depends on the size of the current x - μ^2 grid. It also depends on how many different sets of tables (un-polarised pdfs, polarised pdfs, fragmentation functions, *etc.*) one wants to store. In this respect, QCDNUM is very user-friendly by always gracefully grinding to a halt if it runs out of memory, with a message that tells how large `nwf0` should be.

4.2 Application Program

To illustrate the use of QCDNUM, we present in Figure 5 the listing of a simple application program. For a detailed description of the subroutine calls, and for additional routines not included in the example, we refer to Section 5.

The first step in a QCDNUM based analysis is initialisation (`qcinit`), setting up the x - μ^2 grid (`gxmake`, `gqmake`) and the calculation of the weight tables (`fillwt`). The weights


```

C -----
program example
C -----
implicit double precision (a-h,o-z)
data ityp/1/, iord/3/, nfin/0/          !unpolarised, NNLO, VFNS
data as0/0.364/, r20/2.D0/             !alphas
external func                           !input parton dists
dimension def(-6:6,12)                  !flavor decomposition
data def /
C--  tb  bb  cb  sb  ub  db  g  d  u  s  c  b  t
C--  -6  -5  -4  -3  -2  -1  0  1  2  3  4  5  6
+ 0., 0., 0., 0., 0., -1., 0., 1., 0., 0., 0., 0., 0., !dval
+ 0., 0., 0., 0., -1., 0., 0., 0., 1., 0., 0., 0., 0., !uval
+ 0., 0., 0., -1., 0., 0., 0., 0., 0., 1., 0., 0., 0., !sval
+ 0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., !dbar
+ 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., !ubar
+ 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., !sbar
+ 78*0. /
data xmin/1.D-4/, nxin/100/, iosp/3/    !x grid, splord
dimension qq(2),wt(2)                   !mu2 grid
data qq/2.D0,1.D4/, wt/1.D0,1.D0/, nqin/60/ !mu2 grid
data q2c/3.D0/, q2b/25.D0/, q0/2.0/     !thresh and mu20
data x/1.D-3/, q/1.D3/, qmz2/8315.25D0/ !output scales
C -----
call qcinit(6,' ')                      !initialise
call gxmake(xmin,1,1,nxin,nx,iosp)      !x-grid
call gqmake(qq,wt,2,nqin,nq)           !mu2-grid
call fillwt(ityp,id1,id2,nw)            !calculate weights
call setord(iord)                       !LO, NLO, NNLO
call setalf(as0,r20)                    !input alphas
iqc = iqfrmq(q2c)                       !mu2c
iqb = iqfrmq(q2b)                       !mu2b
call setcbt(nfin,iqc,iqb,0)             !thresholds in the VFNS
iq0 = iqfrmq(q0)                        !starting scale
call evolfg(ityp,func,def,iq0,eps)      !evolve all pdfs
csea = 2.D0*fvalxq(ityp,-4,x,q,0)      !charm sea at x,Q2
asmz = asfunc(qmz2,nfout,ierr)          !alphas(mz2)
end
C -----
double precision function func(id,x)      !momentum density xf(x)
C -----
implicit double precision (a-h,o-z)
if(id.eq.0) func = gluon(x)              !0 = always gluon
if(id.eq.1) func = dvalence(x)          !1 = defined in def
..
if(id.eq.6) func = strangebar(x)        !6 = defined in def
return
end

```

Figure 5: Listing of a QCDNUM application program evolving a complete set of parton densities in the VFNS at NNLO. The array `def` defines the light quark valence ($xq - x\bar{q}$) and anti-quark ($x\bar{q}$) distributions as an input to the evolution. The x dependence of the input densities is coded in the function `func`. After evolution, the pdfs are interpolated to some x and μ^2 and $\alpha_s(m_Z^2)$ is calculated.

depend on the grid definition and the interpolation order so that `fillwt` must be called after the grid has been defined. The weight tables are calculated for LO, NLO and NNLO as well as for all possible flavour settings in the range $3 \leq n_f \leq 6$ so that you do not have to call `fillwt` again when you set or re-set QCDNUM parameters further downstream. Although the weight calculation is fast (typically about 10–20 s) it may become a nuisance in semi-interactive use of QCDNUM so that there is a possibility to dump the weights to disk and read them back in the next QCDNUM run.

In the example code, the weight calculation is followed by setting the perturbative order (`setord`) and the input value of α_s at some renormalisation scale μ_R^2 (`setalf`). The call to `setcbt` sets the VFNS mode and defines the thresholds on the factorisation scale μ_F^2 . All the calls that set evolution parameters are destructive in the sense that they invalidate the parton densities in memory, if any. In this way all QCDNUM results are consistently obtained with the same value of α_s , the same perturbative order, *etc.*

The second step is to evolve the parton densities from input specified at the scale μ_0^2 . It is important to note that QCDNUM evolves parton *momentum* densities $xf(x)$, although all theory in this write-up is expressed in terms of parton *number* densities $f(x)$. The evolution is done by calling the routine `evolfg` which evolves $2n_f + 1$ input parton densities (quarks plus gluon) in the FFNS or VFNS scheme. The routine internally takes care of the proper decomposition of the input quark densities into singlet and non-singlets. In the VFNS the input scale μ_0^2 must lie below the charm threshold μ_c^2 so that, as a consequence, μ_c^2 must lie above the lower boundary of the μ^2 grid.

The flavour composition of each of the input quark densities is given by a table of weights `def(-6:6,12)`. In the example program, six light quark input densities are defined: three valence densities $x(q - \bar{q})$ and three anti-quark densities $x\bar{q}$. This is sufficient input to run evolutions in the VFNS scheme. One is completely free to define the flavour composition of the input quark densities as long as they form a linearly independent set (QCDNUM checks this). Note that the flavours are ordered according to the PDG convention d, u, s, ... and not u, d, s, ... as often is the case in other programs.

The x dependence of these momentum densities at μ_0^2 must be coded for each identifier in an if-then-else block in the function `func`. The sum rules

$$\begin{aligned} \int_0^1 xg(x)dx + \int_0^1 xq_s(x)dx &= 1, \\ \int_0^1 [d(x) - \bar{d}(x)]dx &= 1, \\ \int_0^1 [u(x) - \bar{u}(x)]dx &= 2 \end{aligned} \tag{4.1}$$

cannot be reliably evaluated by QCDNUM since it has no information on the x -dependence of the pdfs below the lowest grid point in x . These sum rules should therefore be built into the parametrisation of the input densities. The evolution does, of course, conserve the sum rules once they are imposed at μ_0^2 . The easiest way to evolve with a symmetric strange sea is to include $xs_v = x(s - \bar{s})$ in the collection of input densities and set it to zero for all x at the input scale μ_0^2 . In the VFNS at LO or NLO, the generated heavy flavour densities $h = (c, b, t)$ are always symmetric ($xh - x\bar{h} = 0$) but this is not true anymore at NNLO, which generates a small asymmetry.

After the parton densities are evolved, the results can be accessed by `fvalxq`. This routine transforms the parton densities from the internal singlet/non-singlet basis to the flavour basis and returns the gluon, a quark, or an anti-quark momentum density, interpolated to x and μ^2 . Also here the flavours d, u, s, ... are indexed according to the PDG convention. The last call in the example program evolves the input value of α_s to the scale m_Z^2 . This evolution is completely stand-alone and does not make use of the μ^2 grid. The function `asfunc` can thus be called at any point after the call to `qcinit`. We refer to Section 5 for more ways to access the QCDNUM results, and for ways to change the renormalisation scale with respect to the factorisation scale.

QCDNUM has an extensive checking mechanism which maintains internal consistency and verifies that all subroutine arguments supplied by the user are within their allowed ranges. Error messages might pop-up unexpectedly when the renormalisation scale is changed with respect to the factorisation scale because the low end of the μ^2 grid may then map onto values of $\mu_R^2 < \Lambda^2$.

Another QCDNUM feature is that $n_f = (4, 5, 6)$ and not $(3, 4, 5)$ at the heavy flavour thresholds μ_h^2 . This implies, first of all, that parton evolution in the VFNS must start from $\mu_0^2 < \mu_c^2$ and not from $\mu_0^2 \leq \mu_c^2$, simply because the number of flavours must be $n_f = 3$ at the starting scale. There is, however, no restriction on the starting (renormalisation) scale of α_s so that it may very well coincide with a flavour threshold, either before or after varying the renormalisation scale with respect to the factorisation scale. If this happens at NNLO, the input value of α_s is assumed to include the discontinuity.

4.3 Validation and Performance

The CPU time that is needed to evolve a pdf on a discrete grid grows quadratically with the number of grid points in x . With linear (quadratic) interpolation the accuracy increases linearly (quadratically) with the number of grid points. It follows that an r -fold gain in accuracy will cost a factor of r^2 in CPU for linear interpolation but only a factor of r for quadratic interpolation. This reduction in cost motivated the inclusion of quadratic splines in QCDNUM.

To investigate the performance of the two interpolation schemes, we compare results from QCDNUM to those from the N -space evolution program PEGASUS [19]. In this comparison a default set of initial distributions [25] is evolved at NNLO from $\mu^2 = 2$ to $\mu^2 = 10^4 \text{ GeV}^2$ with $n_f = 4$ flavours. The dashed curve in the top plot of Figure 6 shows the relative difference $\Delta g/g$ versus x for QCDNUM evolution with linear splines on a single 200 point grid extending down to $x = 10^{-5}$. The accuracy at low x is satisfactory (few permille) but deteriorates rapidly to $\Delta g/g > 2\%$ for $x > 0.35$.

The precision is much improved by evolving on multiple grids (Section 3.3) as shown by the full curve in the top plot of Figure 6. Here the 200 grid points are re-distributed over five sub-grids with lower limits as given in Table 1. For each successive grid the point density is twice that of the previous grid. It is seen from Figure 6 that the precision is now better than 2% for $x < 0.85$.

The dotted curves in Figure 6 (top and bottom) correspond to evolution with quadratic

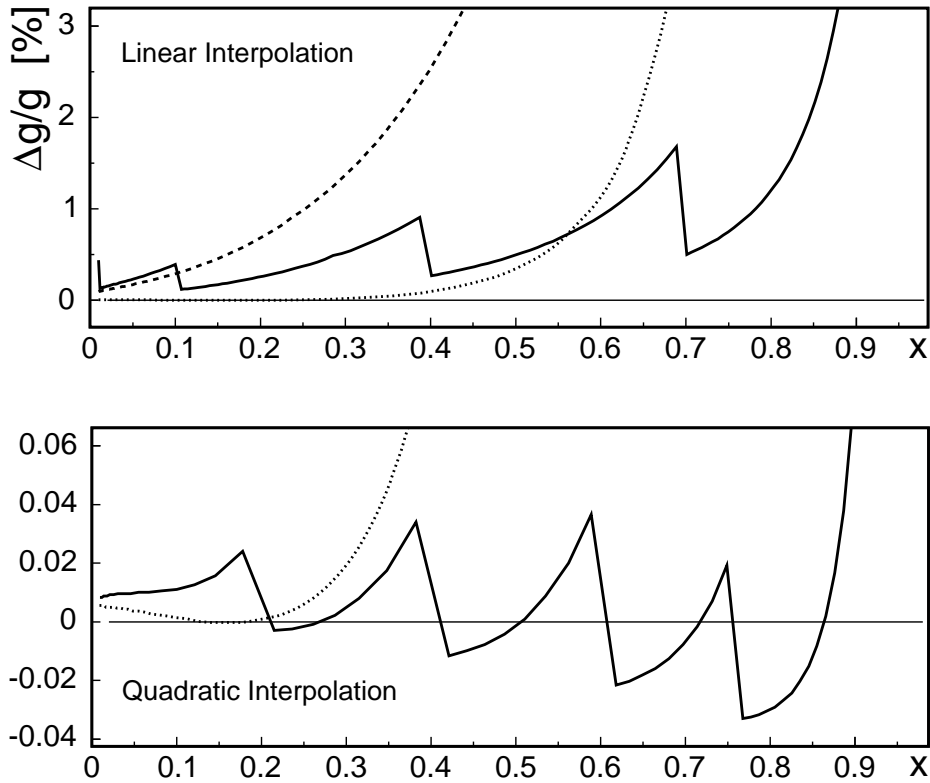


Figure 6: The relative difference $\Delta g/g$ (in percent) of gluon densities evolved from $\mu^2 = 2$ to $\mu^2 = 10^4$ GeV² by QCDNUM and PEGASUS. Top: Evolution with linear splines on a 200 point single grid down to $x = 10^{-5}$ (dashed curve) and on multiple grids (full curve). Bottom: Evolution with quadratic splines on a 100 point single grid (dotted curve, also shown in the top plot) and on multiple grids (full curve). Note the different vertical scales in the two plots.

splines on a single 100 point grid. There is a large improvement in accuracy (more than a factor of 10) compared to linear splines even though the number of grid points is reduced from 200 to 100. However, also here the precision deteriorates with increasing x , reaching a level of 2% at $x = 0.65$. A five-fold multiple grid with lower limits as listed in Table 1 yields a precision $\Delta g/g < 5 \times 10^{-4}$ over the entire range $x < 0.9$ as can be seen from the full curve in the lower plot of Figure 6. Note that this is for evolution up to $\mu^2 = 10^4$ GeV²; at lower μ^2 the accuracy is even better since it increases (roughly linearly) with decreasing $\ln(\mu^2)$. To fully validate the QCDNUM evolution with PEGASUS,¹³ we have made additional comparisons in the FFNS with $n_f = 3, 5$ or 6 flavours, in the VFNS with and without backward evolution, and with the renormalisation scale set different from the factorisation scale. This for both un-polarised evolution up to NNLO and polarised evolution up to NLO.

As remarked in Section 3.3, the quadratic spline evolution is not more expensive in CPU time than linear spline evolution. On the contrary: QCDNUM runs 4 times *faster* since

¹³Similar benchmarking between HOPPET [21] and PEGASUS is given in [25] and [26], where also pdf reference tables can be found. We do not provide here benchmark tables for QCDNUM, but a program that generates such tables and compares them with PEGASUS is available upon request from the author.

Table 1: Lower x limits of multiple grids used in the evolution with linear and quadratic splines.

	n	x_1	x_2	x_3	x_4	x_5
Linear interpolation	200	10^{-5}	0.01	0.10	0.40	0.70
Quadratic interpolation	100	10^{-5}	0.20	0.40	0.60	0.75
Relative point density		1	2	4	8	16

we need only 100 instead of 200 grid points. With the multiple grid definition given in Table 1 for quadratic splines, the density of the first grid ($x > 10^{-5}$) is 12 points per decade. It follows that for evolution down to $x = 10^{-6}$ (10^{-4}) a grid with $100 + 12 = 112$ ($100 - 12 = 88$) points should be sufficient.

To investigate the execution speed we did mimic a QCD fit by performing 1000 NNLO evolutions in the VFNS (13 pdfs), using a 60 point μ^2 grid and the 5-fold 100 point x -grid given in Table 1. After each evolution, the proton structure functions F_2 and F_L were computed at NNLO for 1000 interpolation points in the HERA kinematic range. For this test, QCDNUM, MBUTIL, and ZMSTF were compiled with the GFORTRAN compiler, using level 2 optimisation and without array boundary check. The computations took 18.5 CPU seconds on a 2 GHz Intel Core 2 Duo processor under Mac OS-X: 8.5 s for the evolutions and 10 s for the structure functions.

5 Subroutine Calls

In this section we describe all available QCDNUM subroutines and functions. For convenience a list of these is given in Table 2. In the following we will prefix output variables with an asterisk (*). We use the FORTRAN convention that integer variable and function names start with the letters I–N. Character variables are given in quotes as in 'opt'. Other variables and functions are in double precision unless otherwise stated. Note that floating point numbers should be entered in double precision format:

```
ix = ixfrm ( x )      ! ok
ix = ixfrm ( 0.1D0 )  ! ok
ix = ixfrm ( 0.1 )    ! wrong!
```

Unlike FORTRAN, QCDNUM is case insensitive so that character arguments like 'ALIM' or 'Alim' are both valid inputs.

Most QCDNUM functions will, upon error, generate an error message. The inclusion of function calls in **print** or **write** statements can then cause program hang-up in case the function tries to issue a message. Thus:

```
write(6,*) 'Glue = ', fvalxq(1,0,x,q,1) ! not recommended

glue = fvalxq(1,0,x,q,1)                ! OK
write(6,*) 'Glue = ', glue               ! OK
```

Table 2: Subroutine and function calls in QCDNUM.

Subroutine or function	Description
QCINIT (lun, 'filename')	Initialise
SETLUN (lun, 'filename')	Redirect output
SET GETVAL ('opt', val)	Set Get parameters
SET GETINT ('opt', ival)	Set Get parameters
GXMAKE (xmi, iwt, n, nxin, *nxout, iord)	Define x grid
IXFRMX (x)	Get i_x from x
XFRMIX (ix)	Get x from i_x
XXATIX (x, ix)	Verify grid point
GQMAKE (qarr, wt, n, nqin, *nqout)	Define μ_F^2 grid
IQFRMQ (q2)	Get i_μ from μ_F^2
QFRMIQ (iq)	Get μ_F^2 from i_μ
QQATIQ (q2, iq)	Verify grid point
GRPARS (*nx, *x1, *x2, *nq, *q1, *q2, *io)	Get grid definitions
GXCOPY (*array, n, *nx)	Copy x grid
GQCOPY (*array, n, *nq)	Copy μ^2 grid
FILLWT (itype, *idmi, *idma, *nw)	Fill weight tables
FILLWC (mysub, *idmi, *idma, *nw)	Custom weights
DMPWGT (itype, lun, 'filename')	Dump weight tables
READWT (lun, 'fn', *idmi, *idma, *nw, *ie)	Read weight tables
NWUSED (*nwtot, *nwuse, *nwtab)	Memory words used
SET GETORD (iord)	Set Get order
SET GETALF (alfs, r2)	Set Get α_s start value
SETCBT (nfix, iqc, iqb, iqt)	Set n_f or thresholds
GETCBT (*nfix, *q2c, *q2b, *q2t)	Get n_f or thresholds
SET GETABR (ar, br)	Set Get μ_R^2 scale
RFROMF (fscale)	Convert μ_F^2 to μ_R^2
FFROMR (rscale)	Convert μ_R^2 to μ_F^2
ASFUNC (r2, *nf, *ierr)	Evolve $\alpha_s(\mu_R^2)$
EVOLFG (itype, func, def, iq0, *eps)	Evolve all pdfs
PDFINP (subr, iset, offset, *epsi, *nwds)	Pdfs from outside
CHKPDF (iset)	True if pdf set exists
FVALXQ (iset, id, x, qmu2, ichk)	Interpolate $ g, q, \bar{q}\rangle$
FVALIJ (iset, id, ix, iq, ichk)	$ g, q, \bar{q}\rangle$ at grid point
FPDFXQ (iset, x, qmu2, *pdfs, ichk)	All pdfs $ g, q, \bar{q}\rangle$
FPDFIJ (iset, ix, iq, *pdfs, ichk)	All pdfs $ g, q, \bar{q}\rangle$
FSUMXQ (iset, def, x, qmu2, ichk)	Linear combination
FSUMIJ (iset, def, ix, iq, ichk)	Linear combination
FSNSXQ (iset, id, x, qmu2, ichk)	Interpolate $ g, e^\pm\rangle$
FSNSIJ (iset, id, ix, iq, ichk)	$ g, e^\pm\rangle$ at grid point
FSPLNE (iset, id, x, iq)	Spline interpolation
SPLCHK (iset, id, iq)	Check spline

Output arguments are pre-fixed with an asterisk (*).

5.1 Initialisation

`call QCINIT (lun, 'filename')`

Initialise QCDNUM and define the output stream. Should be called before anything else.

- lun** Output logical unit number. When set to 6, QCDNUM messages appear on the standard output. When set to -6, the QCDNUM banner printout is suppressed on the standard output.
- 'filename'** Output file name. Irrelevant when **lun** is set to 6 or -6.

`call SETLUN (lun, 'filename')`

Redirect the QCDNUM messages. The parameters are as for `qcinit` above. This routine can be called at any time after `qcinit`.

`call SETVAL|GETVAL ('opt', val)`

Set or get QCDNUM floating point parameters.

- 'null'** Result of a calculation that cannot be performed. Default, **null** = 1.D11.
- 'epsi'** The tolerance level in the floating point comparison $|x-y| < \epsilon$, which QCDNUM uses to decide if x and y are equal. Default, **epsi** = 1.D-9.
- 'epsg'** Required numerical accuracy of the Gauss integration in the calculation of weight tables. Default, **epsg** = 1.D-7.
- 'elim'** Allowed difference between a quadratic and a linear spline interpolation mid-between the grid points in x . Default, **elim** = 0.5; larger values may indicate spline oscillation. To disable the check, set **elim** < 0.
- 'alim'** Maximum allowed value of $\alpha_s(\mu^2)$. When α_s exceeds the limit, a fatal error condition is raised. Default, **alim** = 10.¹⁴
- 'qmin'** Smallest possible lower boundary of the μ^2 grid. Default, **qmin** = 0.1 GeV².
- 'qmax'** Largest possible upper boundary of the μ^2 grid. Default, **qmax** = 1.D11 GeV².

These parameters can be set and re-set at any time after `qcinit`.

`call SETINT|GETINT ('opt', ival)`

Set or get QCDNUM integer parameters.

- 'iter'** Set the number of iterations in the backward evolution. When set negative, one will evolve backward in the same interpolation scheme as the forward evolution (not recommended). When set to zero, one will evolve backward in the linear interpolation scheme, without iterations (not recommended either). A value larger than zero gives the number of iterations to perform. Default, **iter** = 1. This parameter is only relevant when one works with quadratic splines.

¹⁴When you raise **alim** > 10 then α_s will at some point be limited by internal cuts in QCDNUM.

- 'lunq' Retrieve the QCDNUM logical unit number. Useful if one wants to write messages on the same output stream as QCDNUM. This option is only available for `getint` and not for `setint`.
- 'ntab' Number of scratch buffers (maximum 20) generated by `fastini` (fast convolution engine, Section 6.5). Will have no effect when set after the call to `fastini`. Default, `ntab = 5`. If one wants to generate more than 20 buffers, the value of `mbf0` in `qcdnum.inc` should be increased, and QCDNUM re-compiled.

5.2 Grid Definition

A proper definition of the grid in x and μ^2 is important because it determines the speed and accuracy of the QCDNUM calculations. The grid definition also governs the partition of the internal store which contains the weight tables and tables of parton densities. In addition, the routines set-up the bases of B-splines.

The x grid must be strictly equidistant in the variable $y = -\ln x$ but in QCDNUM one can generate multiple equidistant grids (Section 3.3) to obtain a finer binning at low y (large x). Multiple grids are generated when the x -range is subdivided into regions with different densities, as is described below.

The μ^2 grid does not need to be equidistant. So one can either enter a fully user-defined grid or let QCDNUM generate one by an equidistant logarithmic fill-in of a given set of intervals in μ^2 .

```
call GXMAKE ( xmin, iwt, n, nxin, *nxout, iord )
```

Generate a logarithmic x -grid.

- xmin** Input array containing `n` values of x in ascending order: `xmin(1)` defines the lower end of the grid while the other values define the approximate positions where the point density will change according to the values set in `iwt`. The list may or may not contain $x = 1$ which is ignored anyway.
- iwt** Input integer weights. The point density between `xmin(1)` and `xmin(2)` will be proportional to `iwt(1)`, that of the next region will be proportional to `iwt(2)` and so on. The weights should be given in ascending order and must always be an integer multiple of the previous weight. Thus, to give an example, the triplets $\{1,1,1\}$ and $\{1,2,4\}$ are allowed but $\{1,2,3\}$ is not.
- n** The number of values specified in `xmin` and `iwt`. This is also the number of sub-grids used internally by QCDNUM.
- nxin** Requested number of grid points (not including the point $x = 1$). Should of course be considerably larger than `n` for an x -grid to make sense.
- nxout** Number of generated grid points. This may differ slightly from `nxin` because of the integer arithmetic used to generate the grid.
- iord** One should set `iord = 2 (3)` for linear (quadratic) spline interpolation.

With this routine, one can define a (logarithmic) grid in x with higher point densities at large x , where the parton distributions are strongly varying. Thus


```

xmin = 1.D-4
iwt = 1
call gxmake(xmin,iwt,1,100,nxout,iord)

```

generates a logarithmic grid with exactly 100 points in the range $10^{-4} \leq x < 1$, while

```

xmin(1) = 1.D-4
iwt(1) = 1
xmin(2) = 0.7D0
iwt(2) = 2
call gxmake(xmin,iwt,2,100,nxout,iord)

```

generates a 100-point grid with twice the point density above $x \approx 0.7$.

A call to `gxmake` invalidates the weight tables and the pdf store.

<code>ix = IXFRMX (x) x = XFRMIX (ix) L = XXATIX (x, ix)</code>

The function `ixfrmx` returns the index of the closest grid point at or below x . Returns zero if x is out of range (note that $x = 1$ is outside the range) or if the grid is not defined. The inverse function is `x = xfrmix(ix)`. Also this function returns zero if `ix` is out of range or if the grid is not defined. To verify that x coincides with a grid point, use the logical function `xxatix`, as in

```

logical xxatix
ix = xfrmix(x)           !x is at or above grid point ix
if(xxatix(x,ix)) then    !x is at grid point ix

```

Note that `QCDNUM` snaps to the grid, that is, x is considered to be at a grid point i if $|y - y_i| < \epsilon$ with $y = -\ln x$ and, by default, $\epsilon = 10^{-9}$.

<code>call GQMAKE (qarr, wgt, n, nqin, *nqout)</code>

Generate a logarithmic μ_F^2 grid on which the parton densities are evolved.¹⁵

- qarr** Input array containing `n` values of μ^2 in ascending order: `qarr(1)` and `qarr(n)` define the lower and upper end of the grid, respectively. The lower end of the grid should be above 0.1 GeV^2 . If `n` > 2 then the additional points specified in `qarr` are put into the grid. In this way, one can incorporate a set of starting values μ_0^2 , or thresholds $\mu_{c,b,t}^2$.
- wgt** Input array giving the relative grid point density in each region defined by `qarr`. The weights are not restricted by integer multiples as in `gxmake` but can be set to any value in the range $0.1 \leq w \leq 10$. With these weights, one can generate a grid with higher density at low values of μ^2 where α_s is changing rapidly.

¹⁵Note that α_s is evolved (without using a grid) on μ_R^2 which may or may not be different from μ_F^2 .

- n** The number of values specified in **qarr** and **wgt** ($n \geq 2$).
- nqin** Requested number of grid points. If $nqin \leq n$ then the grid is not generated but taken from **qarr**. This feature allows you to read-in your own μ^2 grid.
- nqout** Number of generated grid points. This may differ slightly from **nqin** because of the integer arithmetic used to generate the grid.

A call to **gqmake** invalidates the weight tables and the pdf store.

<code>iq = IQFRMQ (q2) q2 = QFRMIQ (iq) L = QQATIQ (q2, iq)</code>
--

The function **iqfrmq** returns the index of the closest grid point at or below μ^2 . The inverse function is **qfrmiq**. To verify that μ^2 coincides with a grid point, use the logical function **qqatiq**. As described above for the corresponding x grid routines, a value of zero is returned if **q2** and/or **iq** are not within the range of the current grid, or if the grid is not defined.

<code>call GRPARS (*nx, *xmi, *xma, *nq, *qmi, *qma, *iord)</code>
--

Returns the current grid definitions

- nx** Number of points in the x grid not including $x = 1$.
- xmi** Lower boundary of the x grid.
- xma** Upper boundary of the x grid. Is always set to **xma** = 1.
- nq** Number of points in the μ^2 grid.
- qmi** Lower boundary of the μ^2 grid.
- qma** Upper boundary of the μ^2 grid.
- iord** Order of the spline interpolation (2 = linear, 3 = quadratic).

<code>call GXCOPY (*array, n, *nx)</code>

Copy the x grid to a local array

- array** Local array containing on exit the x grid but not the value $x = 1$.
- n** Dimension of **array** as declared in the calling routine.
- nx** Number of grid points copied to the local array. A fatal error occurs if **array** is not large enough to contain the current grid.

<code>call GQCOPY (*array, n, *nq)</code>

As above, but now for the μ^2 grid.

5.3 Weights

In this section we describe routines to calculate the weight tables, to dump these to disk and to read them back. The weight tables are calculated for all orders (LO,NLO,NNLO) and all number of flavours $n_f = (3, 4, 5, 6)$, irrespective of the current QCDNUM settings. Tables can be created for un-polarised pdfs, polarised pdfs and fragmentation functions. All these pdf types can exist simultaneously in memory. For each type, one gluon table and 12 quark tables are generated by the routines, in addition to the weight tables.

`call FILLWT (itype, *idmin, *idmax, *nwds)`

Partition the pdf store and fill the weight tables used in the calculation of the convolution integrals. Both the x and μ^2 grid must have been defined before the call to `fillwt`.

itype	Select un-polarised pdfs (1), polarised pdfs (2) or fragmentation functions (3). Any other input value will select un-polarised pdfs (default).
idmin	Returns, on exit, the identifier of the first pdf table. Always <code>idmin = 0</code> .
idmax	Identifier of the last pdf table in the store. Always <code>idmax = 12</code> .
nwds	Total number of words used in memory.

One can create more than one set of tables by calling `fillwt` with different values of `itype`. For instance, the sequence

```
call fillwt(1,idmin,idmax,nw)    !Unpolarised pdfs
call fillwt(2,idmin,idmax,nw)    !Polarised pdfs
```

makes both the un-polarised and the polarised pdfs available. For each pdf type, `fillwt` creates 13 pdf tables. If there is not enough space in memory to hold all the tables, `fillwt` returns with an error message telling how much memory it needs. One should then increase value of `nwf0` in the include file `qcdnum.inc`, and recompile QCDNUM.¹⁶ Note that `fillwt` acts as a do-nothing when the pdf type already exists in memory:

```
call fillwt(1,idmin,idmax,nw)    !Unpolarised pdfs
call fillwt(1,idmin,idmax,nw)    !Do nothing
```

`call DMPWGT(itype, lun, 'filename')`

Dump the weight tables (not the pdf tables) of a given pdf type to disk. When `itype = 0`, all pdf types in memory are dumped.¹⁷ Fatal error if `itype` does not exist. Additional information about the QCDNUM version, grid definition and partition parameters is also dumped, to protect against corruption of the dynamic store when the weights are read back in future QCDNUM runs. The dump is unformatted so that the output file cannot be exchanged across machines.

¹⁶This one may have to repeat several times, because `fillwt` proceeds in stages and is ignorant of the memory requirements of the next stage.

¹⁷ This does not include the weight tables of custom evolution (`itype = 4`, see Section 6.6). Such tables are thus always dumped on a separate file.

<pre>call READWT(lun, 'fname', *idmin, *idmax, *nwds, *ierr)</pre>
--

Read the weight tables from a disk file. Both the x and μ^2 grid must have been defined before the call to `readwt`. On exit the error flag is set as follows:

- 0 Weights are successfully read in.
- 1 Read error or input file does not exist.
- 2 Input file was written with another QCDNUM version.
- 3 Key mismatch (should never occur).
- 4 Incompatible x - μ^2 grid definition.

When successful (`ierr = 0`), the routine creates the pdf store(s) and returns on exit the parameters `idmin`, `idmax` and `nwds` as does the subroutine `fillwt`. One will get a fatal error if there is not enough space in memory to hold all the tables. Like `fillwt`, `readwt` acts as a do-nothing when a pdf type already resides in memory.

The code below automatically maintains an up-to-date weight file on disk:

```
call readwt(lun,'polarised.wgt',idmin,idmax,nw,ierr)
if(ierr.ne.0) then
  call fillwt(2,idmin,idmax,nw)
  call dmpwgt(2,lun,'polarised.wgt')
endif
```

<pre>call NWUSED(*nwtot, *nwuse, *nwtab)</pre>
--

Returns the size `nwtot` of the QCDNUM store (the parameter `nwf0` in `qcdnum.inc`), the number of words used (`nwuse`) and the size of one pdf table (`nwtab`).

5.4 Parameters

In this section we describe the QCDNUM routines to set evolution parameters like the perturbative order, flavour thresholds, α_s , *etc.* All these parameters have reasonable defaults but one can change them at any point in the code. Note that a re-definition of these evolution parameters invalidates the pdf tables of *all* existing types. The weight tables are not invalidated. In this way, all pdfs are always evolved with the same set of parameters; one cannot, for instance, have both un-polarised and polarised pdfs in memory, and evolve one in NNLO and the other in NLO.

<pre>call SETORD GETORD (iord)</pre>
--

Set (or get) the order of the QCDNUM calculations to 1, 2 or 3 for LO, NLO and NNLO, respectively. Default, `iord = 2`.

`call SETALF|GETALF (alfs, r2)`

Set or get for the α_s evolution the starting value `alfs` and the starting renormalisation scale `r2`. Default $\alpha_s(m_Z^2) = 0.118$.

`call SETCBT(nfix, iqc, iqb, iqt)`

nfix Number of flavours in the FFNS mode. If not set to 3, 4, 5 or 6, QCDNUM runs in the VFNS mode.

iqc,b,t Grid indices of the quark mass thresholds $\mu_{c,b,t}^2$. This input is ignored when QCDNUM runs in the FFNS mode, that is, when **nfix** is set to 3, 4, 5 or 6. There are some restrictions, dictated by the evolution and interpolation routines: $iqc \geq 2$, $iqb \geq iqc+2$ and $iqt \geq iqb+2$.

A threshold index value of zero (or larger than the number of grid points) means ‘beyond the upper edge of the grid’. For instance, $(iqc,b,t) = (0,0,0)$ is like running in the FFNS with $n_f = 3$ while the setting $(2,4,0)$ puts the top quark threshold beyond the evolution range. By default, QCDNUM runs in the FFNS with $n_f = 3$.

`call GETCBT(*nfix, *q2c, *q2b, *q2t)`

Return the current threshold settings. If **nfix** is non-zero on return, QCDNUM runs in the FFNS and the values of **q2c,b,t** are irrelevant. When **nfix** = 0, QCDNUM runs in the VFNS and the routine returns the threshold values (not the indices) on the μ^2 scale.

`call SETABR|GETABR (ar, br)`

Define the relation between the factorisation scale μ_F^2 and the renormalisation scale μ_R^2

$$\mu_R^2 = a_R \mu_F^2 + b_R.$$

Default: **ar** = 1 and **br** = 0.

`rscale2 = RFROMF(fscale2) fscale2 = FFROMR(rscale2)`

Convert the factorisation scale μ_F^2 to the renormalisation scale μ_R^2 and *vice versa*.

5.5 Evolution

`alphas = ASFUNC(r2, *nf, *ierr)`

Standalone evolution of α_s on the renormalisation scale μ_R^2 (without using the μ^2 grid or weight tables). QCDNUM internally keeps track of α_s so that there is no need to call this function; it is just a user interface that gives access to $\alpha_s(\mu_R^2)$.

r2 Renormalisation scale μ_R^2 where α_s is to be calculated.
nf Returns, on exit, the number of flavours at the scale **r2**.
ierr = 1 Too low value of **r2**. Internally, there is a cut $\mathbf{r2} > 0.1 \text{ GeV}^2$ and also a cut on the slope, to avoid getting too close to Λ^2 .

The input scale and input value of α_s , the order of the evolution and the flavour thresholds are those set by default or by the routines described in Section 5.4. Note that although α_s is evolved on the renormalisation scale the result, in the VFNS, may still depend on the relation between μ_R^2 and μ_F^2 . This is because the position of the heavy flavour thresholds depends on this relation.

`call EVOLFG(itype, func, def, iq0, *epsi)`

Evolve a complete set of parton *momentum* densities from an input scale μ_0^2 . If QCDNUM runs in the FFNS, the gluon and $2n_f$ quark densities must be given as an input at μ_0^2 . In the VFNS, the gluon and $2n_f = 6$ light quark densities must be given at $\mu_0^2 < \mu_c^2$.

Here and in the following the parton densities are written on the flavour basis (note the PDG convention) with an indexing defined by

$$\begin{array}{cccccccccccccccc}
 -6 & -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
 \hline
 \bar{t} & \bar{b} & \bar{c} & \bar{s} & \bar{u} & \bar{d} & g & d & u & s & c & b & t
 \end{array} \tag{5.1}$$

itype Type of evolution: un-polarised (1), polarised (2), time-like (3), or custom (4).
func User defined function `func(j,x)` (see below) that returns the input parton momentum density $xf_j(x)$ at **iq0**. Must be declared **external** in the calling routine. The index **j** runs from 0 (gluon) to $2n_f$.
def Input array dimensioned in the calling routine to `def(-6:6,12)` which contains in `def(i,j)` the contribution of parton species **i** to the input distribution **j**, that is, `def(i,j)` specifies the flavour decomposition of all input distributions **j**. The indexing of **i** is given in (5.1). Internally, QCDNUM constructs from **def** a $2n_f \times 2n_f$ sub-matrix of coefficients and tries to invert this matrix. If that fails, the $2n_f$ input densities are not linearly independent in flavour space and an error condition is raised, see also (2.32).
iq0 Grid index of the starting value μ_0^2 . When evolving in the FFNS the input scale can be anywhere inside the range of the μ^2 grid. In the VFNS, however, μ_0^2 should be below the charm threshold.
epsi Maximum deviation of the quadratic spline interpolation from linear interpolation mid-between the grid points (see Section 3.3). A large value **epsi** > **elim** may indicate spline oscillation and will cause a fatal error message. The value of **elim** can be set by a call to `setval`. When **elim** ≤ 0 the error condition is disabled so that one can investigate the cause of oscillation. Note that, by definition, **epsi** = 0 when QCDNUM is run in the linear interpolation scheme.

The input function **func** must be coded as follows

```

double precision function func(ipdf,x)
implicit double precision (a-h,o-z)
if(ipdf.eq.0) then
    func = xgluon(x)                !0 = gluon  xg(x)
elseif(ipdf.eq.1) then
    func = my_favourite_quark_dstn_1(x) !1 = quarks xq1(x)
elseif(ipdf.eq.2) then
    func = my_favourite_quark_dstn_2(x) !2 = quarks xq2(x)
elseif(ipdf.eq.3) then
    ..
endif
return
end

```

Because `evolfg` will call `func` only at the grid points x_i , it is possible to feed tabulated values into the evolution routine as is illustrated by the following code

```

double precision function pdfinput(ipdf,x)
implicit double precision (a-h,o-z)
common /input/ table(0:12,nxx)    !table with input values
ix      = ixfrm(x)
pdfinput = table(ipdf,ix)
return
end

```

Here is code that evolves both un-polarised and polarised pdfs.

```

call fillwt(1, idmin, idmax, nw)          !unpolarised
call fillwt(2, idmin, idmax, nw)          !polarised
..
call evolfg(1, func1, def1, iq01, epsi1)  !unpolarised
call evolfg(2, func2, def2, iq02, epsi2)  !polarised

```

5.6 External Pdfs

In QCDNUM, one can read up to 5 different pdf sets from some external source, with type identifiers running from 5 to 9. Before reading an external pdf set, care should be taken that the perturbative order, the flavour scheme, the positions of the thresholds and the input value of α_s are set correctly in QCDNUM. Otherwise one will get the wrong answer when the pdf set is used later on in structure function or cross-section calculations. We remind that *all* pdf sets in memory—including the external ones—are invalidated when a QCDNUM parameter is re-set by calling one of the routines in Section 5.4.

<pre>call PDFINP (subr, iset, offset, *epsi, *nwds)</pre>

subr User supplied subroutine (see below), declared `external` in the calling routine.

iset	Pdf set identifier in the range 5–9. If the pdf set already exists, it will be overwritten.
offset	Relative offset at the thresholds μ_h^2 . This parameter is used to catch discontinuities at the thresholds, if any, by sampling the pdfs at $\mu_h^2(1 \pm \delta)$. A small number like 10^{-3} should be sufficient, but this depends on how the pdfs are externally represented, and how accurate the thresholds are set in QCDNUM.
epsi	Maximum deviation of the quadratic spline interpolation from linear interpolation mid-between the grid points. As for the routine evolsg , a large value epsi > elim may indicate spline oscillation and will cause a fatal error message. Note that, by definition, epsi = 0 when QCDNUM is run in the linear interpolation scheme.
nwds	Last word occupied in the store. Fatal error if the store is not large enough.

The routine **subr** provides the interface between QCDNUM and the external repository:

```

subroutine SUBR ( x, qmu2, xf )
implicit double precision (a-h,o-z)
dimension xf(-6:6)
..

```

The output array **xf**(-6:6) should contain the values of the gluon and the (anti-)quark momentum densities at x and μ^2 , indexed according to (5.1); note the PDG convention.

5.7 Pdf Interpolation

Here we describe routines to access the gluon distribution (xg), the quark and anti-quark distributions ($xq, x\bar{q}$), or linear combinations of the quarks and anti-quarks. It is also possible to directly access the basis singlet/non-singlet pdfs in memory (xe^\pm , defined in Section 2.4). These routines perform local polynomial interpolation on a $k \times 3$ mesh around the interpolation point in x and μ^2 , where k is the current interpolation order in x . Fast routines return the value of a pdf at a given grid point (**ix, iq**). Two routines are provided to investigate the behaviour of the internal spline representation in x .

In the routines below, the pdf set identifier **iset** selects the pdf set (or type): unpolarised (1), polarised (2), fragmentation function (3), custom (4), or external (5–9).

Lval = CHKPDF(iset)

Returns **.true.** if the pdf set exists in memory. Both **Lval** and **chkpdf** should be declared **logical** in the calling routine.

pdf = FVALXQ (iset, id, x, qmu2, ichk)

Returns the gluon density or one of the (anti-)quark densities, interpolated to x and μ^2 .

iset Pdf set identifier [1–9].

id Gluon, quark or anti-quark identifier, indexed as given in (5.1).
x, qmu2 Input value of x and μ^2 .
ichk If set to zero, **fvalxq** will return a null value when **x** or **qmu2** are outside the grid boundaries; if set to a non-zero value a fatal error message will be issued.

The fast version of this function is: `pdf = fvalij(iset, id, ix, iq, ichk)`.

`call FPDFXQ (iset, x, qmu2, *pdfs, ichk)`

Returns all pdf values in one call. The arguments are as given above, except

pdfs Output array, dimensioned to `pdfs(-6:6)` in the calling routine. The indexing is given in (5.1).

The fast version is the subroutine `fpdfij(iset, ix, iq, *pdfs, ichk)`.

`pdf = FSUMXQ (iset, def, x, qmu2, ichk)`

Return a weighted sum of quark densities. The arguments are as given above, except

def Input array, dimensioned to `def(-6:6)` in the calling routine, containing the coefficients of the linear combination. The indexing is as given in (5.1) but note that `def(0)` is ignored since it does not correspond to a quark density.

The fast call is: `pdf = fsumij(iset, def, ix, iq, ichk)`.

`pdf = FSNSXQ (iset, id, x, qmu2, ichk)`

Return the gluon density or one of the singlet/non-singlet basis pdfs. The arguments are as given above, except that **id** is now indexed as follows:

0	1	2	3	4	5	6	7	8	9	10	11	12	
g	q_s	e_2^+	e_3^+	e_4^+	e_5^+	e_6^+	q_v	e_2^-	e_3^-	e_4^-	e_5^-	e_6^-	(5.2)

The fast call is: `pdf = fsnsij(iset, id, ix, iq, ichk)`.

`pdf = FSPLNE (iset, id, x, iq)`

This routine is identical to **fsnsxq** above, except that the local polynomial interpolation in x is replaced by spline interpolation, as is done in the QCDNUM evolution and convolution routines (note that **fsplne** does not interpolate in μ^2). This function is provided as a diagnostic tool to investigate quadratic spline oscillations, if any, which may not be visible in the local polynomial interpolation. You do not need this function to *detect* spline oscillations, since that is done automatically by **evolsg** and **pdfinp**.

`epsi = SPLCHK (iset, id, iq)`

Returns $\epsilon = \|\mathbf{u} - \mathbf{v}\|$ at a grid point **iq**. Here \mathbf{u} and \mathbf{v} are the vectors of quadratic and linear interpolation mid-between the grid points in x , as is described in Section 3.3. By definition, $\epsilon = 0$ for linear interpolation, and should be a small number (like 0.05, say) for quadratic interpolation. Large values indicate that the spline oscillates.

6 Convolution Engine

The QCDNUM convolution engine provides tools to calculate structure functions in deep inelastic scattering, hadron-hadron scattering cross-sections and parton luminosities. The engine drives the add-on package ZMSTF that computes the zero-mass structure functions F_2 , F_L and xF_3 in un-polarised deep inelastic scattering. It is also used in the HQSTF package that computes the heavy flavour contributions to F_2 and F_L in the fixed flavour number scheme [16]. Both these packages are included in the QCDNUM distribution and are described in the Sections C.3 and D.1 of this write-up.

From the parton *number* densities f and kernels K , all kind of convolution integrals can be calculated with the engine, such as

$$x[f \otimes K](x), \quad x[f \otimes K_a \otimes K_b](x), \quad x[f_a \otimes f_b](x), \quad x[f_a \otimes f_b \otimes K](x), \quad \text{etc.}$$

Here \otimes stands for Mellin convolution as defined by (2.5). We refer to Section 3.2 for how convolution integrals are computed and where the factor x in front comes from. We emphasise that the kernel K must be defined by convolution with a *number* density. If not, then it must be transformed as necessary, before it is fed into QCDNUM.

The steps to be taken in a calculation based on the convolution engine are the following.

1. Declare one or more stores and partition these into tables. Then fill the tables with weights for all the convolution kernels needed in the calculation (Section 6.2);
2. Write a function `myfun(ix,iq)` that returns the structure function, cross section or luminosity at a grid point in x and μ^2 (Section 6.3);
3. Pass `myfun` to a QCDNUM routine that will take care of the interpolation to any desired x and μ^2 (Section 6.4).

This procedure is fairly straight-forward and therefore suitable for prototyping and debugging. However, there is a considerable amount of overhead so that it is recommended to ultimately move steps 2 and 3 of the computation to a fast calculation scheme that is described in Section 6.5. By this one will gain at least an order of magnitude in speed.

Before we present the convolution engine we will first, in the next section, introduce the rescaling variable χ to accommodate generalised mass variable flavour number schemes (GM-VFNS, see [27] for a recent review) in structure function calculations.

6.1 Rescaling Variable in Convolution Integrals

The general expression for a structure function can be written as

$$\mathcal{F}_i(x, Q^2) = \sum_j x \int_\chi^1 \frac{dz}{z} f_j(z, \mu^2) C_{ij} \left[\frac{\chi}{z}, \mu^2, Q^2, m_h^2, \alpha_s(\mu^2) \right]. \quad (6.1)$$

Here the index i labels the structure function (*e.g.* F_2 , F_L , xF_3 , F_2^c , ...) and j labels a parton number density like the gluon, the singlet and various non-singlets. The coefficient function C_{ij} depends on x , on the scale variables μ^2 and Q^2 , on one or more quark

masses m_h^2 and on the strong coupling constant α_s . The variable $\chi = ax$, $a \geq 1$, is a so-called *rescaling* variable which takes into account the kinematic constraints of heavy quark production, for instance,

$$\chi = ax = \left(1 + \frac{4m_h^2}{Q^2}\right)x. \quad (6.2)$$

We have $0 \leq \chi \leq 1$ so that the range of x in (6.1) is restricted to $0 \leq x \leq 1/a$. In the zero-mass limit $a = 1$, $\chi = x$, and (6.1) reduces to the Mellin form $x[f \otimes C](x)$.

To calculate the structure function, we first have to evaluate the convolution integrals (for clarity we drop α_s and the indices i, j)

$$\mathcal{F}(x, Q^2) = x \int_{\chi}^1 \frac{dz}{z} f(z, \mu^2) C\left(\frac{\chi}{z}, \mu^2, Q^2, m_h^2\right). \quad (6.3)$$

As in Section 3.2 we denote by $h(y, t)$ a parton momentum density in the logarithmic scaling variables $y = -\ln x$ and $t = \ln \mu^2$. In terms of these, and provided that χ is proportional to x , (6.3) can be written as a weighted sum of spline coefficients

$$\mathcal{F}(y_i, Q^2) = \sum_{j=1}^i W_{ij} A_j \quad (6.4)$$

with $W_{ij} = w_{i-j+1}$ and

$$w_\ell = e^{-b} \int_0^{y_\ell - b} dz Y_1(z) D(y_\ell - b - z, t, Q^2, m_h^2) \quad (1 \leq \ell \leq n). \quad (6.5)$$

Here $D(y, t, Q^2, m_h^2) = e^{-y} C(e^{-y}, e^t, Q^2, m_h^2)$ and $b = \ln(a)$. It is understood that the integral (6.5) is set to zero in case $y_\ell - b \leq 0$. In the massive schemes, $b > 0$ depends on t which implies that the weights must be stored in 2-dimensional y - t tables.

We emphasise that convolution integrals found in the literature must, if necessary, be brought into the general form (6.1) by modifying the published Wilson coefficient. An example of such a modification can be found in Appendix D.

6.2 Weight Tables

In this section we describe routines that partition a linear store into tables and fill these tables with weights used in the calculation of convolution integrals. It is important to realise that the convolution kernels may contain singularities, see also Appendix A. To deal with such singularities, we formally decompose a kernel into a regular part (A), a singular part (B), a product (RS) and a delta function

$$C(x) = A(x) + [B(x)]_+ + R(x)[S(x)]_+ + D(x)\delta(1-x). \quad (6.6)$$

QCDNUM provides routines that can calculate weights for each term separately (if present) and add these to the weight table of C .

For reasons of efficiency and economy of storage, there are four different types of tables:

<code>itype = 1</code>	Weights that depend only on x . Table identifiers run from 101–199;
<code>itype = 2</code>	Weights that depend on x and n_f . Identifiers run from 201–299;
<code>itype = 3</code>	Weights that depend on x and μ^2 . Identifiers run from 301–399;
<code>itype = 4</code>	Weights that depend on x , μ^2 and n_f . Identifiers run from 401–499.

Although it is a good idea to take out as many μ^2 -dependent factors as possible from the convolution kernel, it is clear from (6.3) that quark mass parameters and the relation between μ^2 and Q^2 may enter via the rescaling variable χ and that this dependence can never be factored out of the convolution integral. Thus the weight tables of the GM schemes will, in general, depend on x and μ^2 and must be stored in type-3 or 4 tables.

QCDNUM calculates by Gauss quadrature (CERNLIB routine D103) the integrals that define the weights. In case the default accuracy of $\epsilon = 10^{-7}$ cannot be reached (fatal error message), this limit can be raised by a call to `setval('epsg', value)`. Note, however, that problems with the Gauss integration will most likely be caused by problems with the integrand—such as near-singular behaviour somewhere in the integration domain—and that this cannot be cured by relaxing the required accuracy.

In Table 3 we list all available weight routines.

Table 3: QCDNUM convolution weight table routines.

Subroutine or function	Description
<code>BOOKTAB (w, nw, itypes, *nwords)</code>	Partition into tables
<code>MAKEWTA (w, id, afun, achi)</code>	Regular piece $A(x)$
<code>MAKEWTB (w, id, bfun, achi, nodelta)</code>	Singular piece $[B(x)]_+$
<code>MAKEWRS (w, id, rfun, sfun, achi, nodelta)</code>	Product $R(x)[S(x)]_+$
<code>MAKEWTD (w, id, dfun, achi)</code>	Delta function $D(x)\delta(1-x)$
<code>MAKEWTX (w, id)</code>	Weight table for $x[f_a \otimes f_b]$
<code>SCALEWT (w, c, id)</code>	Scale weight table
<code>IDSPFUN ('pij', iord, itype)</code>	Splitting function index
<code>COPYWGT (w, id1, id2, iadd)</code>	Copy weight table
<code>WCROSSW (w, ida, idb, idc, iadd)</code>	Double convolution weights
<code>WTIMESF (w, fun, id1, id2, iadd)</code>	Multiply by $f(\mu^2, n_f)$
<code>SETWPAR (w, pars, n)</code>	Store extra information
<code>GETWPAR (w, *pars, n)</code>	Read extra information
<code>TABDUMP (w, lun, 'filename', 'key')</code>	Dump to disk
<code>TABREAD (w, n, lun, 'fn', 'key', *nw, *ierr)</code>	Read from disk

Output arguments are pre-fixed with an asterisk (*).

call BOOKTAB (w, nw, itypes, *nwords)

Partition a store `w` into tables.

w	Double precision array declared in the calling routine.
nw	Dimension of w as declared in the calling routine.
itypes	Integer array dimensioned to itypes(4) in the calling routine which contains in itypes(i) the number of tables (≤ 99) of type i to be generated. When itypes(i) = 0 then no tables of type i will be generated.
nwords	Gives, on exit, the number of words used in the store. If nwords is negative, then the store is not sufficiently large and should be re-dimensioned in the calling routine to at least -nwords .

Note that one can declare and partition as many stores as desired, one per structure function for instance.

call MAKEWTA (w, id, afun, achi)

Calculate the weights for the regular contribution $A(x)$ to a convolution kernel and add these to table **id** in the store **w**.

w	Store declared in the calling routine and previously partitioned by booktab .
id	Table identifier. To add results to a type- <i>n</i> table, one should use identifiers in the range n01–n99 , with n = 1, 2, 3 or 4 .
afun	User function (see below) returning the regular piece of the convolution kernel. Should be declared external in the calling routine.
achi	User function (see below), declared external in the calling routine, that returns the value <i>a</i> of the rescaling variable $\chi = ax$.

The function **afun** provides an interface between QCDNUM and the regular part of the kernel $C(\chi, \mu^2, Q^2, m_h^2)$ and should be coded as follows.¹⁸

```
double precision function afun(chi,qmu2,nf) !chi = a*x
implicit double precision (a-h,o-z)
common /fixpar/ par1, par2, ..... !parameters, if any
Q2 = some_function_of(qmu2,some_params) !Q2
afun = cfun(chi,qmu2,Q2,nf,some_params) !convolution kernel
return
end
```

The function **achi** should return, as a function of μ^2 , the factor *a* that defines the rescaling variable $\chi = ax$.

```
double precision function achi(qmu2)
implicit double precision (a-h,o-z)
common /fixpar/ par1, par2, ..... !parameters, if any
Q2 = some_function_of(qmu2,some_params) !Q2
achi = some_function_of(Q2,some_params)
return
end
```

¹⁸We assume here that the kernel conforms to (6.1). If not, then **afun** must take care of this.

QCDNUM insists that always $\text{achi} \geq 1$, one will get a fatal error if not. To compute standard Mellin convolutions $x[f \otimes C](x)$, simply set $\text{achi} = 1$ for all μ^2 .

```
double precision function achi(qmu2)
implicit double precision (a-h,o-z)
achi = 1.D0
return
end
```

<code>call MAKEWTB (w, id, bfun, achi, nodelta)</code>
--

Calculate the weights for the singular contribution $[B(x)]_+$ to a convolution kernel and add these to a table in the store **w**. The arguments and the coding of **bfun** and **achi** are as for **makewta**. Thus, if a kernel has both a regular and a singular part, then do

```
call makewa(w,201,afun,achi)    !put weights in id = 201
call makewb(w,201,bfun,achi,0) !add weights to id = 201
```

It is seen from Appendix A, equation (A.4), that a ‘+’ prescription generates a $\delta(1-x)$ contribution. By default, **makewtb** includes this contribution, unless you set **nodelta** = 1. In that case the $\delta(1-x)$ contribution is not calculated and must be entered, perhaps combined with other such contributions, via a call to **makewtd**, see below.

<code>call MAKEWRS (w, id, rfun, sfun, achi, nodelta)</code>
--

Calculate the weights for the product contribution $R(x)[S(x)]_+$ to a convolution kernel and add these to a table in the store **w**. The arguments and the coding of **rfun**, **sfun** and **achi** are as for **makewta**.

<code>call MAKEWTD (w, id, dfun, achi)</code>

Calculate the weights for the $\delta(1-x)$ contribution to a convolution kernel and add these to a table in the store **w**. The delta function is multiplied by the function **dfun**. The arguments and the coding of **dfun** and **achi** is as for **makewta**.

<code>call MAKEWTX (w, id)</code>

Calculate the weights (3.20) for the convolution $x[f_a \otimes f_b](x)$.

w	Store declared in the calling routine and previously partitioned by booktab .
id	Table identifier. Because the weight table depends only on x , it can be stored in a type-1 table, but equally well in types-2, 3 or 4, if desired.

<code>call SCALEWT (w, c, id)</code>
--

Multiply the contents of table `id` by a constant `c`.

<code>id = IDSPFUN ('pij', iord, itype)</code>
--

Return the index (< 0) of a splitting function weight table stored internally in QCDNUM.

'pij' Name of the splitting function. Valid input strings are

PQQ, PQG, PGQ, PGG, PPL, PMI, PVA.

iord Select LO (1), NLO (2) or NNLO (3).

itype Select evolution type: un-polarised (1), polarised (2), fragmentation function (3) or custom (4).

The index returned by `idspfun` is encoded as $-(1000*itype+id)$, where `id` is the internal table identifier. If the table does not exist, the function returns a value of -1.

<code>call COPYWGT (w, id1, id2, iadd)</code>

Copy the contents of table `id1` to `id2`.

w Store declared in the calling routine.

id1 Input table identifier. One can copy a splitting function weight table from internal QCDNUM memory to the store by setting `id1 < 0`. Valid identifiers are generated by `idspfun()`, as described above.

id2 Output table identifier with `id2 \neq id1`. The output table type may be different from the input table type, see below.

iadd If set to 0 copy `id1` to `id2`, if set to +1 (-1) add (subtract) `id1` to (from) `id2`.

For this routine—and for those described below—the output table type can be different from the input table type, provided that this does not lead to a loss of input information. Thus one can copy a type-1 table to a type-3 table but not the other way around (fatal error). Note that input splitting function weight tables are all type-2.

<code>call WCROSSW (w, ida, idb, idc, iadd)</code>
--

This routine generates a weight table for the convolution of two kernels K_a and K_b . The weight table is calculated with (3.18) from two input tables \mathbf{W}_a and \mathbf{W}_b .

w Store declared in the calling routine.

- ida** Table identifier containing the weights of kernel K_a . When **ida** < 0 one will access a splitting function weight table which is stored internally in QCDNUM. See **idspfun()** above for how to generate a valid splitting function identifier.
- idb** As above for the weights of kernel K_b .
- idc** Output table identifier. Cannot be set equal to **ida** or **idb**.
- iadd** If set to 0 store the result of the convolution in **idc**, if set to +1 (-1) add (subtract) the result to (from) the contents of **idc**.

The table types of **ida** and **idb** may be different, but the type of **idc** must be such that it can contain either input table. Thus if **ida** is type-2 (x, n_f) and **idb** is type-3 (x, μ^2), then **idc** must be type-4 (x, μ^2, n_f). The routine checks this.

<code>call WTIMESF (w, fun, id1, id2, iadd)</code>
--

Multiply a weight table by a function of μ^2 and n_f and store the result in another table.

- w** Store declared in the calling routine.
- fun** User supplied double precision function **fun(iq,nf)** declared **external** in the calling routine.
- id1** Input weight table identifier. It is possible to access a splitting function weight table by setting **id1** < 0. Valid identifiers can be obtained from **idspfun()** described above.
- id2** Identifier of the output table. It is allowed to have **id1** = **id2** (in-place modification of a table), unless **id1** is a splitting function table. The table type of **id2** must be such that no information is lost. The routine checks this.
- iadd** Store the result in **id2** in case **iadd** = 0 or add (subtract) the result to (from) **id2** in case **iadd** = +1 (-1).

The routine loops over **iq** and **nf** and calls **fun(iq,nf)** with the following argument ranges, depending on the output table type:

type	variables	iq range	nf range
1	x	1–1	3–3
2	x, n_f	1–1	3–6
3	x, μ^2	1–nq	3–3
4	x, μ^2, n_f	1–nq	3–6

With this routine one can, in combination with **wcrossw**, construct weight tables for combinations of convolution kernels, such as those given in (C.7). For instance, here is code that generates a table for

$$C_{2,+}^{(2,1)} = C_{2,q}^{(0)} \otimes P_+^{(1)} + C_{2,+}^{(1)} \otimes P_{qq}^{(0)} - \beta_0 C_{2,+}^{(1)}.$$

```

external beta0    !beta function
..
call WcrossW ( w, idC2Q0, idSpfun('PPL',2,1), idC2P21, 0 )
call WcrossW ( w, idC2P1, idSpfun('PQQ',1,1), idC2P21, +1 )
call WtimesF ( w, beta0 , idC2P1                , idC2P21, -1 )

```


<code>call SETWPAR (w, par, n)</code>

Write extra information to the store, for instance quark masses or other parameters that you may want to dump to disk, together with the tables themselves.

w Store, partitioned by a previous call to `booktab`.
par List of parameters to be written. Should be dimensioned to at least `par(n)` in the calling routine.
n Number of items to be written up to a maximum of `miw0 = 20`. If necessary, one can change the value of `miw0` in `qcdnum.inc` and recompile `QCDNUM`.

The parameters can be read back by a call to `getwpar(w,par,n)`.

<code>call TABDUMP (w, lun, 'filename', 'key')</code>

Dump the store `w` to disk. Apart from the store, information is written about the `QCDNUM` version, the x - μ^2 grid definition and the current spline interpolation order. The `key` text string can be used to stamp the file with a version number or other identifier. The dump is unformatted so that the file cannot be exchanged across machines.

<code>call TABREAD (w, nw, lun, 'filename', 'key', *nwords, *ierr)</code>

Read a store from disk into the array `w(nw)`. The size of the store (in words) is returned in `nwords`. You will get a fatal error message if `w(nw)` is not large enough to contain the store. Note that the x and μ^2 grids must have been defined before the call to `tabread`. On exit, the error flag is set as follows (non-zero means that nothing has been read in).

- 0 Store successfully read in.
- 1 Read error or input file does not exist.
- 2 File written by another `QCDNUM` version.
- 3 Key mismatch.
- 4 Incompatible x - μ^2 grid definition.

`QCDNUM` insists that the key written on the file matches the key entered as an argument to `tabread`.¹⁹ Thus if, for instance, the key is set to a package name and version number then the user of the package cannot read obsolete files written by earlier versions, or read files written by another package. If you don't want to use keys, just enter an empty string as a key in the calls to `tabdump` and `tabread`.

¹⁹Note that the key matching is case insensitive and that leading and trailing blanks are ignored.

6.3 Convolution

In Table 4 we list the routines that can be used to build a structure function, cross-

Table 4: Calls in the QCDNUM convolution engine.

Subroutine or function	Description
FCROSSK (w, idw, iset, idf, ix, iq)	Convolution $x[f \otimes K]$
FCROSSF (w, idw, iset, ida, idb, ix, iq)	Convolution $x[f_a \otimes f_b]$
EFROMQQ (qvec, *evec)	Transform from q, \bar{q} to e^\pm
QQFROME (evec, *qvec)	Transform from e^\pm to q, \bar{q}
NFLAVOR (iq)	Returns n_f
GETALFN (iq, n, *ierr)	Returns $(\alpha_s/2\pi)^n$

Output arguments are pre-fixed with an asterisk (*).

section or parton luminosity at a grid point in x and μ^2 .

A convolution is always computed with the pdfs in QCDNUM memory, that is, with the gluon density or with one of the singlet/non-singlet quark densities $|e^\pm\rangle$ as defined in Section 2.4. To translate a linear combination of quarks and anti-quarks to the $|e^\pm\rangle$ basis, and *vice versa*, the routines **efromqq** and **qqfrome** are provided.

For convenience we show here again the indexing (5.2) of the singlet/non-singlet basis

$$\begin{array}{cccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \hline g & q_s & e_2^+ & e_3^+ & e_4^+ & e_5^+ & e_6^+ & q_v & e_2^- & e_3^- & e_4^- & e_5^- & e_6^- \end{array}, \quad (6.7)$$

and the indexing (5.1) of the flavour basis

$$\begin{array}{cccccccccccccc} -6 & -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \bar{t} & \bar{b} & \bar{c} & \bar{s} & \bar{u} & \bar{d} & g & d & u & s & c & b & t \end{array}. \quad (6.8)$$

```
val = FCROSSK ( w, idw, iset, idf, ix, iq )
```

Calculate the convolution $x[f \otimes K](x)$ at a grid point in x and μ^2 .

w Store declared in the calling routine and previously filled with weights.
idw Identifier of a table in the store **w**.
iset Pdf set identifier [1–9].²⁰
idf Pdf identifier, indexed according to (6.7).
ix, iq Indices of an $x\text{-}\mu^2$ grid point.

Splitting function tables cannot be directly accessed by this routine; they should first be copied to the store by a call to **copywgt**.

²⁰ Un-polarised (1), polarised (2), fragmentation function (3), custom (4), or external (5–9).

```
val = FCROSSF ( w, idw, iset, ida, idb, ix, iq )
```

Calculate the convolution $x[f_a \otimes f_b](x)$ at a grid point in x and μ^2 .

w Store declared in the calling routine and previously partitioned by **booktab**.
idw Identifier of a weight table, previously filled by a call to **makewx**.
iset Pdf set identifier [1–9].
ida, idb Pdf identifiers, indexed according to (6.7).
ix, iq Indices of an $x\text{--}\mu^2$ grid point.

A convolution of a linear combination of pdfs must be calculated as a sum of pair-wise convolutions, with each term computed by **fcrossf**. Note that this is much easier done with the fast routines described in Section 6.5.

```
call EFROMQQ ( qvec, *evec, nf )
```

Transform the coefficients of a linear combination of quarks and anti-quarks from the flavour basis to the singlet/non-singlet basis as described in Section 2.4.

qvec Input array, dimensioned **qvec**(–6:6), filled with the coefficients of a linear combination of quarks and anti-quarks and indexed according to (6.8).
evec Output array, dimensioned to **evec**(12), filled with the coefficients written on the singlet/non-singlet basis, indexed according to (6.7).
nf Active number of flavours. This parameter is needed to construct the appropriate $2n_f \times 2n_f$ transformation matrix that acts on **qvec**(–**nf**:**nf**).

Thus if a linear combination of quarks and anti-quarks is written as

$$|p\rangle = \sum_{i=1}^{n_f} (\alpha_i |q_i\rangle + \beta_i |\bar{q}_i\rangle) = \sum_{i=1}^{n_f} (d_i^+ |e_i^+\rangle + d_i^- |e_i^-\rangle) \quad (6.9)$$

and α_i and β_i are stored in the input vector **qvec**, then the coefficients d_i^\pm are returned in the output vector **evec**.

```
call QQFROME ( evec, *qvec, nf )
```

Transform the coefficients of a linear combination of basis vectors from the singlet/non-singlet basis to the flavour basis. The arguments are as for **efromqq**.

```
nf = NFLAVOR ( iq )
```

Returns the number of active flavours at the grid point **iq**. Note that this number is (4,5,6) and not (3,4,5) at the thresholds (**iqc**,**iqb**,**iqt**).

<code>as = GETALFN (iq, n, *ierr)</code>
--

Returns the value of $(\alpha_s/2\pi)^n$ at the factorisation scale μ_F^2 . Here α_s is computed from the Taylor expansion (2.17), appropriately truncated depending on the current value of the perturbative order (`iord`). Because the truncation is different for the pdfs (Section 2.3) and the structure functions (Appendix C.2), the value of `n` must be set as follows.

- If the convolution at (LO, NLO, NNLO) should be multiplied by $(\alpha_s, \alpha_s^2, \alpha_s^3)$ then you set `n = (1,2,3)` in the call to `getalfn`.
- If the convolution at (LO, NLO, NNLO) should be multiplied by $(1, \alpha_s, \alpha_s^2)$ then you set `n = (0,-1,-2)` in the call to `getalfn`.
- If `n > iord`, the value of $(\alpha_s/2\pi)^n$ is calculated at μ_R^2 , instead of at μ_F^2 . (This is already the case for `n = iord`, see Section 2.3.)

To have access to the NNLO discontinuities at the thresholds, one can set `iq` positive (includes discontinuity) or negative (does not include discontinuity), thus:

```
call getalfn ( iqcharm, n, ierr )      !result for nf = 4
call getalfn ( -iqcharm, n, ierr )    !result for nf = 3
```

In other words, by preceding `iq` with a minus sign one effectively changes the QCDNUM default $n_f = (4, 5, 6)$ at the thresholds to the alternative $n_f = (3, 4, 5)$. When `iq` is close to or below the value of Λ^2 , then `ierr = 1` and `getalfn` returns the `null` value. This also happens when `iq` is outside the grid boundaries (`ierr = 2`).

Note that the QCDNUM expansion parameter is $\alpha_s/2\pi$ but that many convolution kernels found in the literature are defined for an expansion in $\alpha_s/4\pi$, in which case one must account for the appropriate factors of 2^n somewhere in the calculation.

6.4 Interpolation

With the routines presented above one can write a function `stfun(ix,iq)` that returns the value of a structure function or cross section at a grid point in x and μ^2 . The routine `stfunxq` then takes care of the interpolation to any value of x and μ^2 . This interpolation is done on a $k \times 3$ mesh around the interpolation point, where k is set to the current spline interpolation order ($2 = \text{linear}$, $3 = \text{quadratic}$). Thus $3k$ functions have to be computed for each interpolation which becomes inefficient if there are interpolations with overlapping meshes. By processing lists of interpolation points, instead of each point individually, redundant calculations are avoided which can lead to considerable gains in computing time. In other words, `stfunxq` should not be called in a loop over interpolation points but be given the *list* of points.

<code>call STFUNKQ (stfun, x, qmu2, stf, n, ichk)</code>
--

Interpolate the function `stfun(ix,iq)` to a list of x and μ^2 values.

stfun	Double precision function, declared external in the calling routine, that returns the value of a structure function or cross-section at $(\mathbf{x}, \mathbf{iq})$.
x, qmu2	List of interpolation points, dimensioned to at least n in the calling routine.
stf	Contains, on exit, the list of interpolated results.
n	Number of items in x , qmu2 and stf .
ichk	If set to 0 the routine returns a null value if x or μ^2 are outside the boundaries of the grid; if set non-zero it will insist that all interpolation points are inside the grid boundaries.

Note that the interpolation is done in μ^2 and not in Q^2 . You have to keep track yourself of the relation between these two scales.

6.5 Fast Computation

As already remarked above, the routines provided up to now are fine for prototyping but are slow because there is quite a lot of overhead when the calculation is repeated at more than one interpolation point. Here we describe a set of routines that does optimised bulk calculations on selected points in the x - μ^2 grid. With these fast routines one can easily gain one or two orders of magnitude in speed. To make the calculation flexible it is broken down into small steps where intermediate results are stored into scratch buffers (by default, the fast engine generates 5 scratch buffers but one can have more, if necessary). An optimised calculation proceeds as follows.

1. Pass a list of interpolation points in x and μ^2 to a QCDNUM routine that determines which grid points will be occupied in the course of the calculation;
2. Store a pdf or a linear combination of pdfs in a scratch buffer;
3. Convolve the pdf with a convolution kernel or a perturbative expansion of kernels;
4. Multiply the convolution by a function of x and μ^2 , for instance by some power of α_s or by some kinematic factor;
5. Accumulate the cross section or structure function in a final buffer;
6. Pass this buffer to an interpolation routine to get a list of interpolated results.

The list of subroutines is given in Table 5. In principle, the output buffer of any fast routine can serve as the input buffer of any other fast routine. There is, however, a little complication related to the amount of information stored in a buffer. For interpolation purposes, it is sufficient to store results only at the mesh points; such a buffer is called *sparse*. A convolution routine, on the other hand, does not only need the values at the mesh points x_i , but also the values at all points $x_j > x_i$. An input buffer with such a storage pattern is called *dense*; a dense buffer is of course more expensive to generate than a sparse buffer. Usually one does not have to worry about sparse and dense buffers, because QCDNUM has reasonable defaults on what kind of buffer is accepted as input,

Table 5: Fast convolution engine.

Subroutine or function	Description
FASTINI (<i>x</i> , <i>qmu2</i> , <i>n</i> , <i>ichk</i>)	Pass list of x and μ^2 values
FASTCLR (<i>id</i>)	Clear buffer
FASTEPM (<i>iset</i> , <i>idf</i> , <i>idout</i>)	Store $ g, e^\pm\rangle$ in a scratch buffer
FASTSNS (<i>iset</i> , <i>pdf</i> , <i>isel</i> , <i>idout</i>)	Store singlet/non-singlet component
FASTSUM (<i>iset</i> , <i>coef</i> , <i>idout</i>)	Store weighted sum of $ e^\pm\rangle$
FASTFXK (<i>w</i> , <i>idw</i> , <i>idf</i> , <i>idout</i>)	Convolution $x[f \otimes K](x)$
FASTFXF (<i>w</i> , <i>idw</i> , <i>ida</i> , <i>idb</i> , <i>idout</i>)	Convolution $x[f_a \otimes f_b](x)$
FASTKIN (<i>id</i> , <i>fun</i>)	Scale by a kinematic factor
FASTCPY (<i>idin</i> , <i>idout</i> , <i>iadd</i>)	Copy or accumulate result
FASTFXQ (<i>id</i> , <i>*f</i> , <i>n</i>)	Interpolation

Output arguments are pre-fixed with an asterisk (*).

and what kind of buffer is generated on output. One can always override the output default and force a routine to generate a dense or a sparse buffer, as needed.

The pdf set parameter *iset* in the routines **fastepm**, **fastsns** and **fastsum** selects the pdf set, namely, un-polarised (1), polarised (2), fragmentation function (3), custom (4), or external (5–9).

call FASTINI (*x*, *qmu2*, *n*, *ichk*)

Pass a list of interpolation points and, at the first call, generate the set of scratch buffers.

x Array, dimensioned to at least *n* in the calling routine, filled with x values.
qmu2 As above, but for μ^2 (not Q^2).
n Number of entries in **x** and **qmu2**.
ichk If non-zero, **fastini** insists that all x and μ^2 are within the grid boundaries.

By default, 5 scratch buffers (*id* = 1–5) are generated at the first call (or cleared if they exist). This number can be changed by calling **setint('ntab',ival)** prior to **fastini**. One will get a fatal error if there is not enough space for the scratch buffers, in which case one has to increase the value of **nwf0** in **qcdnum.inc**, and recompile **QCDNUM**.

call FASTCLR (*id*)

Clear a scratch buffer. Setting *id* = 0 will clear all buffers.

call FASTEPM (*iset*, *idf*, *idout*)

Copy the gluon density or one of the basis pdfs $|e^\pm\rangle$ to a scratch table.

iset Input pdf set identifier [1–9].
idf Pdf identifier [0–12], indexed according to (6.7).
idout Output scratch table identifier [1–5].

By default, **fastepm** generates a dense buffer; a sparse buffer is generated when you pre-pend the output identifier with a minus sign.

```

call fastEpm(1, 0, 1)      !dense table output
call fastEpm(1, 0, -1)     !sparse table output

```

call FASTSNS (iset, pdf, isel, idout)
--

Decompose a given linear combination of quarks and anti-quarks into singlet and non-singlet components and copy a specific component to a scratch buffer.

iset Input pdf set identifier [1–9].
pdf Input array, dimensioned **pdf**(–6:6), filled with the coefficients of a linear combination of quarks and anti-quarks and indexed according to (6.8).
isel Selection flag [0–7], see below.
idout Output scratch table identifier [1–5].

The **isel** flag selects the gluon density (0), the singlet component q_s (1), the non-singlet component q_{ns}^+ (2), the valence component q_v (3), the non-singlet component q_{ns}^- (4), the sum $q_v + q_{ns}^-$ (5), all non-singlets $q_v + q_{ns}^- + q_{ns}^+$ (6) or all quarks (7).

Note that the singlet is weighted by an appropriate n_f dependent factor, for instance by the average square of the quark charges in case **pdf** corresponds to the charge weighted sum of quarks and anti-quarks. Note also that the gluon density is multiplied by the same factor, as is required in structure function calculations, see Appendix C. Setting **isel** = 0 or 1 is thus not the same as calling **fastepm** for the identifiers 0 or 1.

By default, **fastsns** generates a dense buffer; a sparse buffer is generated when you pre-pend the output identifier with a minus sign.

call FASTSUM (iset, coef, idout)

Copy a linear combination of basis pdfs $|e^\pm\rangle$ to a scratch table.

iset Input pdf set identifier [1–9].
coef Array of coefficients dimensioned **coef**(0:12,3:6) in the calling routine.
idout Output scratch table identifier [1–5].

The array **coef**(i,nf) is indexed according to (6.7). Here is code that fills **coef** by transforming a set of quark coefficients from flavour space to singlet/non-singlet space:

```

dimension qvec(-6:6), coef(0:12,3:6)
do nf = 3,6
  coef(0,nf) = 0.DO                                !gluon coefficient
  call efromqq(qvec, coef(1,nf), nf)                !quark coefficients
enddo

```

By masking out coefficients, one can copy the singlet component, or various combinations of non-singlets; this is exactly what **fastsns** does. To copy the gluon distribution, one must set all coefficients to zero, except **coef(0,nf)**.

By default, **fastsum** generates a dense buffer; a sparse buffer is generated when you pre-pend the output identifier with a minus sign.

```
call FASTFXK ( w, idw, idf, idout )
```

Calculate the convolution $x[f \otimes K](x)$ at all selected grid points.

w	Store, declared in the calling routine and previously filled with weights.
idw	Set of weight identifiers, declared idw(4) in the calling routine, see below.
idf	Input scratch buffer, previously filled by fastepm , fastsns or fastsum .
idout	Output scratch table with idout \neq idf .

One can either convolve with a given weight table or with a perturbative expansion of weight tables, depending on what one puts in the array **idw**:

1. To convolve with a given weight table, set **idw(1)** to the identifier of that weight table and set **idw(2)**, **idw(3)** and **idw(4)** to zero;
2. To convolve with a perturbative expansion, store the (LO,NLO,NNLO) weight table identifiers in **idw(1)**, **idw(2)** and **idw(3)**. Set the identifier to zero if no such table exists, as is the case for F_L at LO, for instance. Declare in **idw(4)** the leading power of α_s , that is, multiply (LO,NLO,NNLO) by $(1, \alpha_s, \alpha_s^2)$ if **idw(4)** = 0 and by $(\alpha_s, \alpha_s^2, \alpha_s^3)$ if **idw(4)** = 1. Note that the perturbative expansion is summed up to the current perturbative order, as defined by an upstream call to **setord**.

The routine only accepts a dense buffer as input (otherwise fatal error) and will, by default, generate a sparse buffer as output. If you pre-pend the output identifier with a minus sign, the output buffer will be dense. In this way, the output table can serve as an input to another convolution which allows one to calculate multiple convolutions in a chain. For example,²¹

```

call fastSum( 1, coef, 1 )      ! 1 = f
call fastFxK( w, idK1, 1, -2 ) ! 2 = f * K1
call fastFxK( w, idK2, 2, 3 )  ! 3 = f * K1 * K2

```

²¹It is more efficient, however, to first calculate with **wcrossw** a weight table for $K_3 = K_1 \otimes K_2$, and use that table to convolve K_3 with f .

<code>call FASTFXF (w, idx, ida, idb, idout)</code>

Calculate the convolution $x[f_a \otimes f_b](x)$ at all selected grid points.

w Store, declared in the calling routine and previously partitioned by `booktab`.
idx Identifier of a weight table, previously filled by a call to `makewtx`.
ida, idb Identifiers of input scratch tables. It is allowed to have `ida = idb`.
idout Output scratch table with `idout` \neq `ida` or `idb`.

As above, the routine accepts only dense buffers as input, and generates a sparse buffer as output, unless the output identifier is pre-pended by a minus sign, thus,

```

call fastSum( 1, coefa, 1 )           ! 1 = fa
call fastSum( 1, coefb, 2 )           ! 2 = fb
call fastFxF( w, idwX, 1, 2, -3 )     ! 3 = fa * fb
call fastFxK( w, idwK, 3, 4 )         ! 4 = fa * fb * K

```

<code>call FASTKIN (id, fun)</code>

Multiply the contents of a scratch table by a kinematic factor.

id Identifier of the input scratch table.
fun Double precision function, declared `external` in the calling routine.

The routine loops over the selected grid points and calls the user supplied function `fun` that should return the kinematic factor. The syntax of `fun` is

```
double precision function fun ( ix, iq, nf, ithresh )
```

ix,iq Grid point indices.
nf Number of flavors at `iq`. This number is bi-valued at the thresholds so that at the charm threshold, for instance, `nf` can be either 3 or 4.²²
ithresh Set to 0 if `iq` is not at a threshold and to +1 (-1) if `iq` is at a threshold with the upper (lower) number of flavours. This variable can be used to take NNLO discontinuities into account, as is shown in the example below.

```

double precision function fkin(ix,iq,nf,ithresh)
..
if(ithresh.ge.0) then
  alfas = getalfn( iq,1,ierr)      !alfas/2pi with discontinuity
else
  alfas = getalfn(-iq,1,ierr)      !without discontinuity
endif
..

```

²² The reader may wonder when QCDNUM returns the value 3, and when the value 4. This depends on the interpolation point μ^2 to which `iq` is associated: if μ^2 is below (above) μ_c^2 , then `nf` = 3 (4).

<code>call FASTCPY (idin, idout, iadd)</code>

Copy or accumulate a result in an output buffer.

idin Identifier of the input scratch table.
idout Identifier of the output scratch table with **idout** \neq **idin**.
iadd Store (0), add (1) or subtract (-1) the result to **idout**.

The type of output buffer (sparse or dense) is the same as that of the input buffer, except that once you have used a sparse input buffer, the output buffer will be flagged a sparse and will remain so until you set **iadd** = 0 to start a new accumulation in **idout**.

<code>call FASTFXQ (id, *f, n)</code>

Interpolate the contents of **id** to the list of x and μ^2 values that was previously passed to QCDNUM by the call to **fastini**.

id Identifier of an input scratch buffer.
f Array dimensioned to at least **n** in the calling routine that will contain, on exit, the interpolated values.
n Number of interpolations requested.

The routine works through the list of interpolation points given in the call to **fastini** and exits when it reaches the end of that list or when the number of interpolations is equal to **n**, whatever happens first. A dense input buffer is allowed, but wasteful since it contains a lot of information that is not used by **fastfxq**.

6.6 Custom Evolution

In Section 5.3 we have described the **fillwt** routine to generate weight tables (and pdf tables) for the evolution of un-polarised pdfs (**itype** = 1), polarised pdfs (2), and fragmentation functions (3). But QCDNUM can handle yet another type of evolution, with user-defined evolution kernels (custom evolution, **itype** = 4). For this one has to provide a subroutine, described below, that generates the weight tables. This routine is passed to QCDNUM as an argument of the custom weight filling routine **fillwc**, after which the custom evolution becomes available by switching to **itype** = 4.

```
external mysub, func
..
call fillwc( mysub, idmin, idmax, nwords )
..
call evolfg( 4, func, def, iq0, epsi )
..
```

```

C      -----
      subroutine myweight(w,nw,nwords,idpij,mxord,idum)
C      -----

C--   w          (in)   qcdnum store passed by reference
C--   nw          (in)   number of words available
C--   nwords      (out)  number of words used < 0 not enough space
C--   idpij       (out)  list of Pij table identifiers
C--   mxord       (out)  maximum perturbative order LO,NLO,NNLO
C--   idum        (out)  not used at present

      implicit double precision (a-h,o-z)

      dimension w(*), idpij(7,3), itypes(4)

      external AChi
      external PQQR, PQQS, PQQD                      ! PQQ
      external PQGA, PGQA                             ! PQG, PGQ
      external PGGA, PGGR, PGGS, PGGD                ! PGG

      call setUmsg('myweight')    !s/r name for error messages
C-1   Max perturbative order
      mxord = 1
C-2   Partition
      itypes(1) = 2
      itypes(2) = 2
      itypes(3) = 0
      itypes(4) = 0
      call BookTab(w,nw,itypes,nwords)
C-3   Not enough space
      if(nwords.le.0) return
C-4   Assign table indices
      idPij(1,1) = 101                      ! PQQ
      idPij(2,1) = 201                      ! PQG
      idPij(3,1) = 102                      ! PGQ
      idPij(4,1) = 202                      ! PGG
      idPij(5,1) = 101                      ! PPL
      idPij(6,1) = 101                      ! PMI
      idPij(7,1) = 101                      ! PVA
C-5   Fill tables
      call MakeWRS(w, idPij(1,1), PQQR, PQQS, AChi, 0)
      call MakeWtD(w, idPij(1,1), PQQD, AChi)
      call MakeWtA(w, idPij(2,1), PQGA, AChi)
      call MakeWtA(w, idPij(3,1), PGQA, AChi)
      call MakeWtA(w, idPij(4,1), PGGA, AChi)
      call MakeWRS(w, idPij(4,1), PGGR, PGGS, AChi, 0)
      call MakeWtD(w, idPij(4,1), PGGD, AChi)
C--   Done!
      call clrUmsg          !clear s/r name for error messages
      return
      end

```

Figure 7: Subroutine that generates weight tables of user-given evolution kernels (LO only, in this example). The subroutine is passed to QCDNUM via the routine `fillwc`, as is described in the text.

In Figure 7, we show the listing of a custom weight routine (called `myweight`) that creates the weight tables of, in fact, un-polarised splitting functions in LO, see Appendix A. The arguments of such a subroutine are as follows.

```
subroutine mysub( w, nw, nwords, idpij, mxord, idum )
implicit double precision (a-h,o-z)
dimension w(*), idPij(7,3)
..
```

w The QCDNUM store (passed by reference);
nw The number of words available in the store (input, passed by QCDNUM);
nwords Number of words used by the tables (output);
idpij List of weight table identifiers (output);
mxord Maximum perturbative order supported by the tables (output);
idum Not used at present.

The body of the code in Figure 7 shows the steps to be taken in a custom weight routine.

1. Set the maximum order of the custom evolution, here `mxord = 1`;
2. Partition the store into tables by a call to `booktab`. Here are booked two type-1 and two type-2 tables;
3. Branch-out if there is not enough space in the store, as is signalled by a negative value of `nwords` returned by `booktab`;
4. Return in `idPij(id,iord)` the identifiers of the various splitting function tables. The first index runs as follows

1	2	3	4	5	6	7
P_{qq}	P_{qg}	P_{gq}	P_{gg}	P_+	P_-	P_v

There are only LO splitting functions in the example, with the non-singlet splitting functions all being equal to P_{qq} (table identifier 101);

5. Generate the weight tables by calls to `makewta`, `makewtb`, `makewrs`, and `makewd`, as is described in Section 6.2. The calls in Figure 7 actually accommodate the splitting functions given in (A.3).

One can have only one set of custom weight tables in memory; a second call to `fillwc` will result in a fatal error message.

A custom evolution must obey the DGLAP evolution equations (2.9) and (2.12) which implies that the evolution must properly split into singlet/gluon and non-singlet parts. We remind the reader that the evolution kernels in QCDNUM are defined by convolution with parton *number* densities, and not parton momentum densities. These kernels can depend on x , n_f and μ^2 and can be stored in tables of types-1, 2, 3 or 4. Note, however, that the μ^2 dependence via α_s is taken care of in the evolution routine `evolsg`, through multiplication of the N^ℓLO tables by $(\alpha_s/2\pi)^{\ell+1}$.

6.7 Error Messages in Add-On Packages

When one writes an add-on package, the problem arises that QCDNUM error messages will be labelled with the name of the QCDNUM routine and not with that of the package routine. This can of course become confusing for the user who should be aware only of the package routines, and not what is inside.

One solution is that the package catches errors before QCDNUM does, but this would duplicate a good checking mechanism which is already in place. An easier solution is to pass a string to QCDNUM which contains the name of the package routine so that it will be printed together with the error message. For this, the routines `setUmsg` and `clrUmsg` are provided. For instance one of the first calls in the `zmfillw` routine of the ZMSTF package is

```
call setUmsg('ZMFILLW')
```

so that, upon error, the user gets additional information:

```
-----  
Error in BOOKTAB ( W, NW, ITYPES, NT, NWDS ) ---> STOP  
-----  
No x-grid available  
Please call GXMAKE  
  
BOOKTAB was called by ZMFILLW
```

The last call in `zmfillw` is

```
call clrUmsg
```

that wipes the additional message. This is important because downstream QCDNUM errors would otherwise appear to have always come from `zmfillw`.

7 Acknowledgements

I am of course indebted to the original authors of QCDNUM, in particular to M. Virchaux who introduced me to the program in 1991.²³ I thank M. Cooper-Sarkar for using preliminary versions of QCDNUM17 in her QCD fits and providing important feedback during the development phase of the present version. I greatly benefited from the many clarifying discussions with A. Vogt and thank him for the code of the NNLO splitting and coefficient functions. I am grateful to him and to M. Cooper-Sarkar, E. Laenen and R. Thorne for comments on the manuscript. This work is part of the research programme of the Foundation for Fundamental Research on Matter (FOM), which is financially supported by the Netherlands Organisation for Scientific Research (NWO).

²³It was sad to hear that Marc Virchaux passed away in November 2004.

A Singularities

In this appendix we denote by $f(x)$ a parton *momentum* density and not a number density. In terms of f the convolution integrals in the evolution equations read

$$I(x) = \int_x^1 dz P(z) f\left(\frac{x}{z}\right). \quad (\text{A.1})$$

The LO splitting matrix $P_{ij}^{(0)}$ in (2.14) is written as, in the notation of (2.7),

$$\begin{pmatrix} P_{qq}^{(0)} & P_{qg}^{(0)} \\ P_{gq}^{(0)} & P_{gg}^{(0)} \end{pmatrix} = \begin{pmatrix} P_{qq}^{(0)} & 2n_f P_{qig}^{(0)} \\ P_{gqi}^{(0)} & P_{gg}^{(0)} \end{pmatrix}. \quad (\text{A.2})$$

The LO un-polarised splitting functions are given by

$$\begin{aligned} P_{qq}^{(0)}(x) &= \frac{4}{3} \left[\frac{1+x^2}{(1-x)_+} + \frac{3}{2} \delta(1-x) \right] \\ P_{qig}^{(0)}(x) &= \frac{1}{2} [x^2 + (1-x)^2] \\ P_{gqi}^{(0)}(x) &= \frac{4}{3} \left[\frac{1+(1-x)^2}{x} \right] \\ P_{gg}^{(0)}(x) &= 6 \left[\frac{x}{(1-x)_+} + \frac{1-x}{x} + x(1-x) + \left(\frac{11}{12} - \frac{n_f}{18} \right) \delta(1-x) \right]. \end{aligned} \quad (\text{A.3})$$

For the time-like evolution of fragmentation functions, the splitting functions $P_{qig}^{(0)}$ and $P_{gqi}^{(0)}$ are exchanged in (A.2) [15]. The ‘+’ prescription in (A.3) is defined by

$$[f(x)]_+ = f(x) - \delta(1-x) \int_0^1 f(z) dz \quad (\text{A.4})$$

so that

$$\int_x^1 f(z) [g(z)]_+ dz = \int_x^1 [f(z) - f(1)] g(z) dz - f(1) \int_0^x g(z) dz. \quad (\text{A.5})$$

For reference we give the expressions for I_{qq} and I_{gg} obtained from (A.3) and (A.4)

$$\begin{aligned} I_{qq}^{(0)}(x) &= \frac{4}{3} \int_x^1 dz \frac{1}{1-z} \left[(1+z^2) f\left(\frac{x}{z}\right) - 2f(x) \right] + \frac{4}{3} f(x) \left[\frac{3}{2} + 2 \ln(1-x) \right] \\ I_{gg}^{(0)}(x) &= 6 \int_x^1 dz \frac{1}{1-z} \left[z f\left(\frac{x}{z}\right) - f(x) \right] + 6 \int_x^1 dz \left[\frac{1-z}{z} + z(1-z) \right] f\left(\frac{x}{z}\right) + \\ &\quad 6 f(x) \left[\ln(1-x) + \frac{11}{12} - \frac{n_f}{18} \right]. \end{aligned} \quad (\text{A.6})$$

To write down a generic expression we decompose a splitting (or coefficient) function into a regular part (A), singular part (B), product of the two (RS) and a delta function

$$P(x) = A(x) + [B(x)]_+ + R(x)[S(x)]_+ + K(x)\delta(1-x) \quad (\text{A.7})$$

where, of course, not all terms have to be present. The following functions are defined in the logarithmic scaling variable $y = -\ln(x)$:

$$h(y) = f(e^{-y}), \quad Q(y) = e^{-y}P(e^{-y}), \quad \bar{A}(y) = e^{-y}A(e^{-y}) \quad (\text{A.8})$$

with similar definitions for \bar{B} and \bar{S} ; however, $\bar{R}(y) = R(e^{-y})$ and $\bar{K}(y) = K(e^{-y})$ without a factor e^{-y} in front. With these definitions (A.1) can be written as

$$\begin{aligned} I(y) &= \int_0^y du \, Q(u) \, h(y-u) = I_1(y) + I_2(y) + I_3(y) + I_4(y) \quad \text{with} \\ I_1(y) &= \int_0^y du \, \bar{A}(u) \, h(y-u); \\ I_2(y) &= \int_0^y du \, \bar{B}(u) \, [h(y-u) - h(y)] - h(y) \int_0^x dz \, B(z); \\ I_3(y) &= \int_0^y du \, \bar{S}(u) \, [\bar{R}(u)h(y-u) - \bar{R}(0)h(y)] - \bar{R}(0)h(y) \int_0^x dz \, S(z); \\ I_4(y) &= \bar{K}(y)h(y). \end{aligned} \quad (\text{A.9})$$

where the last integrals of I_2 and I_3 are still expressed in the variable $x = \exp(-y)$ to avoid integration extending to infinity in our expressions. Note that we are free to swap the arguments u and $y-u$ in (A.9).

B Triangular Systems in the DGLAP Evolution

For the non-singlet evolution we have to solve the equation (see Section 3.3)

$$\mathbf{V}\mathbf{a} = \mathbf{b}. \quad (\text{B.1})$$

The matrix \mathbf{V} is a lower triangular Toeplitz matrix, that is, a matrix with the elements V_{ij} depending only on the difference $i-j$ as is shown in the 4×4 example (3.15). This matrix is uniquely determined by storing the first column in a one-dimensional vector \mathbf{v} so that $V_{ij} = v_{i-j+1}$ for $i \geq j$, and zero otherwise. Eq. (B.1) is, like any other lower triangular system, iteratively solved by forward substitution

$$\begin{aligned} a_1 &= b_1/v_1 \\ a_i &= \frac{1}{v_1} \left[b_i - \sum_{j=1}^{i-1} v_{(i-j+1)} a_j \right] \quad \text{for } i \geq 2. \end{aligned} \quad (\text{B.2})$$

There is no recursion relation between a_{i-1} and a_i so that in each iteration the sums must be accumulated, giving an operation count of $n(n+1)/2$ for a system of n equations. This is as expensive (or cheap) as multiplying the triangular matrix by a vector.

The substitution algorithm can be extended to solve the coupled singlet-gluon equation

$$\begin{pmatrix} \mathbf{V}_{\text{qq}} & \mathbf{V}_{\text{qg}} \\ \mathbf{V}_{\text{gq}} & \mathbf{V}_{\text{gg}} \end{pmatrix} \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{a} & \mathbf{b} \\ \mathbf{c} & \mathbf{d} \end{pmatrix} \begin{pmatrix} \mathbf{f} \\ \mathbf{g} \end{pmatrix} = \begin{pmatrix} \mathbf{r} \\ \mathbf{s} \end{pmatrix}, \quad (\text{B.3})$$

where \mathbf{a} is a short-hand notation for \mathbf{V}_{qq} , *etc.* These matrices are all lower triangular $n \times n$ Toeplitz matrices. Writing out this equation in components it is easy to see that for the first elements f_1 and g_1 we have to solve the 2×2 matrix equation

$$\begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \begin{pmatrix} r_1 \\ s_1 \end{pmatrix} \rightarrow \begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \frac{1}{a_1 d_1 - b_1 c_1} \begin{pmatrix} d_1 & -b_1 \\ -c_1 & a_1 \end{pmatrix} \begin{pmatrix} r_1 \\ s_1 \end{pmatrix}. \quad (\text{B.4})$$

For $i \geq 2$ we have to accumulate the sums

$$\begin{aligned} R_i &= r_i - \sum_{j=1}^{i-1} [a_{(i+1-j)} f_j + b_{(i+1-j)} g_j] \\ S_i &= s_i - \sum_{j=1}^{i-1} [c_{(i+1-j)} f_j + d_{(i+1-j)} g_j] \end{aligned} \quad (\text{B.5})$$

and solve, for each i , the equations

$$\begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \begin{pmatrix} f_i \\ g_i \end{pmatrix} = \begin{pmatrix} R_i \\ S_i \end{pmatrix} \quad (\text{B.6})$$

The operation count of this algorithm is four times that of (B.2), plus some little overhead to solve the 2×2 matrix equations for each i .

C Zero Mass Structure Functions

C.1 General Formalism

The zero-mass structure functions $F_2(x, Q^2)$, $F_L(x, Q^2)$ and $xF_3(x, Q^2)$ in un-polarised deep inelastic scattering are calculated from (6.1) with $\chi = x$. The Wilson coefficients are functions of x (and sometimes n_f) only. We set, for the moment, the physical scale Q^2 equal to the factorisation and renormalisation scale μ^2 and write the singlet/gluon contribution to F_2 and F_L as (there is no contribution to xF_3 since this structure function is a pure non-singlet)

$$\frac{1}{x} \mathcal{F}_i^{(\text{s})}(x, Q^2) = [C_{i,\text{s}} \otimes q_{\text{s}}](x, \mu^2) + [C_{i,\text{g}} \otimes g](x, \mu^2) \quad i = 2, \text{L}. \quad (\text{C.1})$$

Likewise, non-singlet contributions to the structure functions are given by

$$\frac{1}{x} \mathcal{F}_i^{(\text{ns})}(x, Q^2) = [C_{i,\text{ns}} \otimes q_{\text{ns}}](x, \mu^2) \quad i = 2, \text{L}, 3 \quad (\text{C.2})$$

where the label ‘ns’ stands for the non-singlet indices ‘+’, ‘−’ and ‘v’ as defined by (2.11). To be precise on notation: $\mathcal{F}_2 = F_2$, $\mathcal{F}_L = F_L$ and $\mathcal{F}_3 = xF_3$ in (C.1) and (C.2). A structure function is calculated by adding the singlet/gluon and non-singlet parts, weighted by the appropriate combination of electroweak couplings; we refer to [28] for how to compute neutral and charged current cross sections and structure functions in deep inelastic charged lepton and neutrino scattering.

Like the splitting functions, the coefficient functions are expanded in powers of α_s ,

$$C_{i,j}^{\text{N}^\ell\text{LO}} = \sum_{k=0}^{\ell} a_s^k C_{i,j}^{(k)} \quad i = 2, \text{L}, 3 \quad j = \text{g}, \text{s}, +, -, \text{v} \quad (\text{C.3})$$

where $\ell = (0, 1, 2)$ denotes (LO, NLO, NNLO) and $a_s = \alpha_s/2\pi$. The LO coefficient functions are either zero or trivial delta functions:

$$\begin{aligned} C_{2,\text{g}}^{(0)} &= 0 & C_{2,\text{s}}^{(0)} &= \delta(1-x) & C_{2,\text{ns}}^{(0)} &= \delta(1-x) \\ C_{\text{L},\text{g}}^{(0)} &= 0 & C_{\text{L},\text{s}}^{(0)} &= 0 & C_{\text{L},\text{ns}}^{(0)} &= 0 \\ C_{3,\text{g}}^{(0)} &= 0 & C_{3,\text{s}}^{(0)} &= 0 & C_{3,\text{ns}}^{(0)} &= \delta(1-x). \end{aligned} \quad (\text{C.4})$$

The NLO coefficient functions can be found in [9]. For those at NNLO we refer to [29, 30, 31, 32] and the parametrisations given in [33] and [34].

The LO coefficient functions for F_{L} are zero so that the longitudinal structure function vanishes at LO. An alternative, which we call F'_{L} , is calculated from the expansion

$$C_{\text{L},j}^{\text{N}^\ell\text{LO}} = \sum_{k=1}^{\ell+1} a_s^k C_{\text{L},j}^{(k)}. \quad (\text{C.5})$$

In this way, $C_{\text{L},j}^{(1)}$ is used already at LO (giving a non-zero F_{L}) and $C_{\text{L},j}^{(2)}$ at NLO. At NNLO the 3-loop coefficient function $C_{\text{L},j}^{(3)}$ is taken from [35]. As stated in [35], this 3-loop calculation applies only to electromagnetic current exchange so that Z^0 or W^\pm contributions to F'_{L} at NNLO are, at present, not available.

C.2 Renormalisation and Factorisation Scale Dependence

To calculate the *renormalisation* scale dependence ($\mu_{\text{R}}^2 \neq \mu_{\text{F}}^2$) we replace, in the expansion of the coefficient functions, the powers of a_s by the Taylor series given in (2.17). If the expansion (C.3) is used, the truncation of the right-hand side of (2.17) is to order a_s in NLO and a_s^2 in NNLO. If, for F'_{L} , the expansion (C.5) is used, the truncation is to order a_s in LO, a_s^2 in NLO and a_s^3 in NNLO, like for the splitting functions.

To calculate the *factorisation* scale dependence ($Q^2 \neq \mu_{\text{F}}^2$), the coefficient functions in (C.3) and (C.5) are replaced by [33, 34]

$$C_{i,j}^{(0)} \rightarrow C_{i,j}^{(0)} \quad \text{and} \quad C_{i,j}^{(k)} \rightarrow C_{i,j}^{(k)} + \sum_{m=1}^k C_{i,j}^{(k,m)} L_{\text{F}}^m \quad k \geq 1, \quad (\text{C.6})$$

where $L_{\text{F}} = \ln(Q^2/\mu_{\text{F}}^2)$ and $\mu_{\text{F}}^2 = \mu_{\text{R}}^2$. To write compact expressions for the $C_{i,j}^{(k,m)}$, we introduce the following vector notation. In the non-singlet sector we have a one-dimensional vector $\mathbf{C}_i = C_{i,\text{ns}}$ and a 1×1 matrix $\mathbf{P} = P_{\text{ns}}$. In the singlet/gluon sector we have a 2-dimensional row-vector and a 2×2 matrix that are given by

$$\mathbf{C}_i = (C_{i,\text{s}} \ C_{i,\text{g}}) \quad \text{and} \quad \mathbf{P} = \begin{pmatrix} P_{\text{qq}} & P_{\text{qg}} \\ P_{\text{gq}} & P_{\text{gg}} \end{pmatrix}.$$

In this vector notation, the functions $C_{i,j}^{(k,m)}$ in (C.6) are written as

$$\begin{aligned}
C_i^{(1,1)} &= C_i^{(0)} \otimes P^{(0)} \\
C_i^{(2,1)} &= C_i^{(0)} \otimes P^{(1)} + C_i^{(1)} \otimes [P^{(0)} - \beta_0 I] \\
C_i^{(2,2)} &= \frac{1}{2} C_i^{(1,1)} \otimes [P^{(0)} - \beta_0 I] \\
C_i^{(3,1)} &= C_i^{(0)} \otimes P^{(2)} + C_i^{(1)} \otimes [P^{(1)} - \beta_1 I] + C_i^{(2)} \otimes [P^{(0)} - 2\beta_0 I] \\
C_i^{(3,2)} &= \frac{1}{2} \left\{ C_i^{(1,1)} \otimes [P^{(1)} - \beta_1 I] + C_i^{(2,1)} \otimes [P^{(0)} - 2\beta_0 I] \right\} \\
C_i^{(3,3)} &= \frac{1}{3} C_i^{(2,2)} \otimes [P^{(0)} - 2\beta_0 I].
\end{aligned} \tag{C.7}$$

For F_2 , F_L and xF_3 , the coefficients are calculated up to $C_i^{(2,2)}$. For F'_L , on the other hand, all coefficients in (C.7) are computed. Note, however, that quite some convolutions are trivial because the LO coefficient functions are either zero or δ -functions, see (C.4).

As mentioned above, the expression (C.6) applies only when $\mu_F^2 = \mu_R^2$. It is therefore not possible to vary both scales μ_R^2 and Q^2 at the same time.

C.3 The ZMSTF Package

The ZMSTF package is a QCDNUM add-on with routines that calculate the structure functions F_2 , F_L and xF_3 in un-polarised deep inelastic scattering. The structure functions are computed as a convolution of the parton densities with zero-mass coefficient functions, using the convolution engine described in Section 6.

The list of subroutines is given in Table 6. Note that error messages are, in most

Table 6: Subroutine and function calls in ZMSTF.

Subroutine or function	Description
ZMFILLW (*nwords)	Fill weight tables
ZMDUMPW (lun, 'filename')	Dump weight tables
ZMREADW (lun, 'filename', *nwords, *ierr)	Read weight tables
ZMDEFQ2 (a, b)	Define Q^2
ZMABVAL (*a, *b)	Retrieve a and b coefficients
ZMQFRMU (qmu2)	Convert μ_F^2 to Q^2
ZMUFRMQ (Q2)	Convert Q^2 to μ_F^2
ZSWITCH (iset)	Switch pdf set
ZMSTFUN (istf, def, x, Q2, *f, n, ichk)	Structure functions

Output arguments are pre-fixed with an asterisk (*).

cases, issued by the underlying QCDNUM routines and not by the ZMSTF routine itself. However, the calling ZMSTF routine is mentioned in the error message so that you know where it came from.

<code>call ZMFILLW (*nwords)</code>

Fill the weight tables. The tables are calculated for all flavours $3 \leq n_f \leq 6$ and for all orders LO, NLO, NNLO. On exit, the number of words occupied by the store is returned in **nwords**. If you get an error message that the internal store is too small to contain the weight tables, you should increase the value of the parameter **nzmstor** in the include file **zmstf.inc** and recompile ZMSTF.

This routine (or **zmreadw** below) should be called after an $x\text{-}\mu^2$ grid is defined in QCDNUM and before the first call to **zmstfun**.

<code>call ZMDUMPW (lun, 'filename')</code>

Dump the weights in memory via logical unit number **lun** to a disk file. The dump is unformatted so that the weight file cannot be exchanged across machines.

<code>call ZMREADW (lun, 'filename', *nwords, *ierr)</code>

Read weights from a disk file via logical unit number **lun**. On exit, **nwords** contains the number of words read into the store (fatal error if not enough space, see above) and the flag **ierr** is set as follows.

- 0 Weights are successfully read in.
- 1 Read error or input file does not exist.
- 2 Incompatible QCDNUM version.
- 3 Incompatible ZMSTF version.
- 4 Incompatible $x\text{-}\mu^2$ grid definition.

These errors will not generate a program abort so that one should check the value of **ierr**, and take the appropriate action if it is non-zero.

<code>call ZMDEFQ2 (a, b)</code>

Define the relation between the factorisation scale μ_F^2 and Q^2

$$Q^2 = a\mu_F^2 + b.$$

The Q^2 scale can only be varied when the renormalisation and factorisation scales are set equal in QCDNUM. The default setting is **a** = 1 and **b** = 0. The ranges are limited to $0.1 \leq a \leq 10$ and $-100 \leq b \leq 100$.

A call to **zmabval(a,b)** reads the coefficients back from memory. To convert between the scales use:

```

Q2    = zmqfrmu(qmu2)
qmu2  = zmufmq(Q2)

```

call ZSWITCH (iset)

By default, the structure functions are calculated from the un-polarised parton densities, evolved with QCDNUM (`iset` = 1). With this routine one can switch to the custom evolution (4), or to one of the external pdf sets (5–9). Switching to polarised pdfs (2) or to fragmentation functions (3) does not make sense and will produce an error message.

call ZMSTFUN (istf, def, x, Q2, *f, n, ichk)

Calculate a structure function for a linear combination of parton densities.

istf Structure function index (1,2,3,4) = (F_L, F_2, xF_3, F'_L).

def(-6:6) Coefficients of the quark linear combination for which the structure function is to be calculated. The indexing of **def** is given in (5.1).

x, Q2 Input arrays containing a list of x and Q^2 (not μ^2) values.

f Output array containing the list of structure functions.

n Number of items in **x**, **Q2** and **f**.

ichk If set to zero, **zmstfun** will return a **null** value when x or μ^2 are outside the grid boundaries; otherwise you will get a fatal error message. A μ^2 point that is close or below the QCD scale Λ^2 is considered to be outside the grid boundary.

To calculate a structure function for more than one interpolation point, it is recommended to not execute **zmstfun** in a loop but to pass the entire list of interpolation points in a single call. The loop is then internally optimised for greater speed.

D Heavy Quark Structure Functions

A NLO calculation of the heavy quark contributions to the F_2 and F_L structure functions in deep inelastic charged lepton-proton scattering is given in [16]. Only electromagnetic exchange contributions are taken into account. In this calculation, a heavy flavour h is not taken to be a constituent of the incoming proton but is, instead, assumed to be exclusively produced in the hard scattering process. Quarks with pole mass $m < m_h$ are taken to be mass-less so that the input light quark densities should have been evolved in the FFNS with $n_f = (3, 4, 5)$ for $h = (c, b, t)$ [36].

A heavy flavour contribution to F_2 or F_L is calculated from

$$F_k^h(x, Q^2) = \frac{\alpha_s}{2\pi} \left\{ e_h^2 g \otimes \mathcal{C}_{k,g}^{(0)} + \frac{\alpha_s}{2\pi} \left(e_h^2 g \otimes \mathcal{C}_{k,g}^{(1)} + e_h^2 q_s \otimes \mathcal{C}_{k,q}^{(1)} + q_p \otimes \mathcal{D}_{k,q}^{(1)} \right) \right\}, \quad (\text{D.1})$$

where e_h is the charge of the heavy quark (in units of the positron charge), g is the gluon density, q_s is the singlet density and

$$q_p = \sum_{i=1}^{n_f} e_i^2 (q_i + \bar{q}_i)$$

is the charge-weighted proton quark distribution for n_f light flavours. The first term in (D.1) is the LO contribution from the photon-gluon fusion process $\gamma^* g \rightarrow h\bar{h}$. The last three terms correspond to the NLO sub-process $\gamma^* g \rightarrow h\bar{h}g$ and $\gamma^* q \rightarrow h\bar{h}q$.²⁴ For the heavy quark coefficient functions \mathcal{C} and \mathcal{D} in (D.1) we refer to [16].²⁵

In terms of a number density $f(x, \mu^2)$, the convolution integrals in (D.1) are defined by

$$f \otimes \mathcal{C} = \int_{ax}^1 \frac{dz}{z} z f(z, \mu^2) \mathcal{C}(x/z, Q^2, \mu^2, m_h^2) \quad (\text{D.2})$$

where $a = 1 + 4m_h^2/Q^2$ and μ^2 is the factorisation (equals renormalisation) scale which is usually set to $\mu^2 = Q^2$ or $\mu^2 = Q^2 + 4m_h^2$. The kinematic domain where the heavy quarks contribute is restricted by the requirement that the square of the γ^*p centre of mass energy must be sufficient to produce the $h\bar{h}$ pair: $W^2 = M^2 + Q^2(1-x)/x \geq M^2 + 4m_h^2$ so that the lower integration limit $ax \leq 1$ in (D.2). It turns out that the dependence of the coefficient functions on the relation between Q^2 and μ^2 cannot be factorised so that each setting of the scale parameters needs its own set of weight tables. To calculate the renormalisation scale dependence, the powers of $a_s = \alpha_s/2\pi$ in (D.1) are replaced by the Fourier expansion (2.17), truncated to a_s in LO, and to a_s^2 in NLO. Note that one can vary either μ_R^2 or Q^2 with respect to μ_F^2 , but not both at the same time.

The convolution integral (D.2) is not of the general form (6.1): (i) the factor x in front is missing; (ii) the pdf is $xf(x)$ and not $f(x)$ and (iii) the argument of C is x/z and not χ/z . This mismatch is cured by presenting to QCDNUM the modified kernel

$$\tilde{C}(\chi, \mu^2, Q^2, m_h^2) \equiv \frac{a}{\chi} C\left(\frac{\chi}{a}, \mu^2, Q^2, m_h^2\right), \text{ with } \chi \equiv ax.$$

To make the heavy quark calculation available in QCDNUM17 (as it was in QCDNUM16) we provide the add-on package HQSTF described below.

D.1 The HQSTF Package

The HQSTF package calculates up to NLO the heavy flavour contributions to the F_2 or F_L structure functions from pdfs evolved in the FFNS scheme with n_f light flavours. The list of subroutines is given in Table 7. We will only describe here the routines `hqfillw` and `hqstfun`, the other ones being similar to those in the ZMSTF package.

```
call HQFILLW ( istf, qmass, aq, bq, *nwords )
```

Fill the weight tables. To be called before anything else.

istf Select structure function: 1 = F_L , 2 = F_2 and 3 = both.

²⁴In the LO and the first two NLO terms the virtual photon couples to the heavy quark, hence the factor e_h^2 in (D.1). The last NLO term describes the process where the virtual photon couples to a light quark which subsequently branches into a $h\bar{h}$ pair via an intermediate gluon: hence the appearance of the charge weighted sum, q_p , of light quark distributions.

²⁵ Some of these coefficient functions are given as interpolation tables (taken from code provided by S. Riemersma) since they are too complex to be cast into analytical form. Note that in [16] the coefficient functions are convolved with parton momentum densities and not with number densities.

Table 7: Subroutine and function calls in HQSTF.

Subroutine or function	Description
HQFILLW (istf, qmass, aq, bq, *nwords)	Fill weight tables
HQDUMPW (lun, 'filename')	Dump weight tables
HQREADW (lun, 'filename', *nw, *ierr)	Read weight tables
HQPARMS (*qmass, *aq, *bq)	Retrieve parameters
HQQFRMU (qmu2)	Convert μ_F^2 to Q^2
HQMUFRQ (Q2)	Convert Q^2 to μ_F^2
HSWITCH (iset)	Switch pdf set
HQSTFUN (istf, icbt, def, x, Q2, *f, n, ichk)	Structure functions

Output arguments are pre-fixed with an asterisk (*).

qmass(3) Input array with the c, b, t quark masses in GeV. If a quark mass is set to $m_h < 1$ GeV, no tables will be generated for that quark.

aq, bq Defines the relation $Q^2 = a\mu_F^2 + b$.

nwords Gives, on exit, the number of words used in the store.

One will get a fatal error if the store is not large enough to hold all tables. In that case you can increase the value of **nhqstor** in the include file **hqstf.inc** and recompile HQSTF. The values of the mass and scale parameters can be retrieved at any time after the call to **hqfillw** (or **hqreadw**) by a call to **hqparms(qmass,aq,bq)**.

call HQSTFUN (istf, icbt, def, x, Q2, *f, n, ichk)

Calculate the heavy quark contribution to a structure function.

istf Calculate F_L (1) or F_2 (2).

icbt Select contribution from charm (1), bottom (2) or top (3).

def(-6:6) Coefficients of the quark linear combination for which the structure function is to be calculated. The indexing of **def** is given in (6.8).

x, Q2 Input arrays containing a list of x and Q^2 (not μ^2) values.

f Output array containing the list of structure functions.

n Number of items in **x**, **Q2** and **f**.

ichk If set to zero, **hqstfun** will return a **null** value when x or μ^2 are outside the grid boundaries; otherwise one will get a fatal error message.

The routine checks that for **icbt** = (1,2,3) = (c,b,t) the pdfs were evolved in the FFNS with $n_f = (3, 4, 5)$ flavours. When **icbt** is pre-pended by a minus sign, the check on the FFNS remains active but that on the number of flavours is switched off.

Here is a snippet of code that, in combination with **ZMSTF**, calculates the d,u,s contribution, the charm contribution and the total F_2 (neglecting bottom and top) in charged lepton-proton scattering (the pdfs should have been evolved with $n_f = 3$ flavours).

```

dimension x(100),Q2(100),F2dus(100),F2c(100),F2p(100)
dimension proton(-6:6)
data proton /4.,1.,4.,1.,4.,1.,0.,1.,4.,1.,4.,1.,4./ !divide by 9
..
call zmstfun(2, proton, x, Q2, F2dus, 100, ichk)
call hqstfun(2, 1, proton, x, Q2, F2c, 100, ichk)
do i = 1,100
    F2p(i) = F2dus(i) + F2c(i)
enddo

```

References

- [1] V.N. Gribov and L.N. Lipatov, Sov. J. Nucl. Phys. **15**, 438 (1972);
L.N. Lipatov, Sov. J. Nucl. Phys. **20**, 94 (1975);
G. Altarelli and G. Parisi, Nucl. Phys. **B126**, 298 (1977);
Y. Dokshitzer, Sov. Phys. JETP **46**, 641 (1977).
- [2] S. Moch, J.A.M. Vermaseren and A. Vogt, Nucl. Phys. **B688**, 101 (2004),
hep-ph/0403192.
- [3] A. Vogt, S. Moch and J.A.M. Vermaseren, Nucl. Phys. **B691**, 129 (2004),
hep-ph/0404111.
- [4] A. Ouraou, Ph. D. Thesis, Université de Paris-XI (1988);
M. Virchaux, Ph. D. Thesis, Université de Paris-VII (1988).
- [5] M. Virchaux and A. Milsztajn, Phys. Lett. **B274**, 221 (1992).
- [6] NMC, M. Arneodo et al., Phys. Lett. **B309**, 222 (1993).
- [7] ZEUS Collab., M. Derrick et al., Phys. Lett. **B345**, 576 (1995);
ZEUS Collab., J. Breitweg et al., Eur. Phys. J. **C7**, 609 (1999);
ZEUS Collab., S. Chekanov et al., Phys. Rev. **D67**, 012007 (2003).
- [8] M. Botje, Eur. Phys. J. **C14**, 285 (2000).
- [9] W. Furmanski and R. Petronzio, Z. Phys. **C11**, 293 (1982).
- [10] O.V. Tarasov, A.A. Vladimirov and A.Yu Sharkov, Phys. Lett. **B93**, 429 (1980);
S.A. Larin and J.A.M. Vermaseren, Phys. Lett. **B303**, 224 (1993).
- [11] K.G. Chetyrkin, B.A. Kniehl and M. Steinhauser, Phys. Rev. Lett. **79**, 2184 (1997),
hep-ph/9706430.
- [12] G. Gurci, W. Furmanski and R. Petronzio, Nucl. Phys. **B175**, 27 (1980).
- [13] W. Furmanski and R. Petronzio, Phys. Lett. **97B**, 437 (1980).

- [14] R. Mertig and W.L. van Neerven, Z. Phys. **C70**, 637 (1996) hep-ph/9506451;
W. Vogelsang, Nucl. Phys. **B475**, 47 (1996), hep-ph/9603366.
- [15] P. Nason and B.R. Webber, Nucl. Phys. **B421**, 473 (1994); Erratum Nucl.
Phys. **B480**, 755 (1996).
- [16] E. Laenen et al., Nucl. Phys. **B392**, 162 (1993);
S. Riemersma et al., Phys. Lett. **B347**, 143 (1995).
- [17] W.K. Tung et al., J. Phys. **G28**, 983 (2002);
S. Kretzer et al., Phys. Rev. **D69**, 114005 (2004).
- [18] R.S. Thorne, Phys. Rev. **D73**, 054019 (2006), hep-ph/0601245 and references
therein.
- [19] A. Vogt, Comput. Phys. Commun. **170**, 65 (2005), hep-ph/0408244.
- [20] M. Buza et al., Eur. Phys. J. **C1**, 301 (1998), hep-ph/9612398.
- [21] G.P. Salam and J. Rojo, Comput. Phys. Commun. **180**, 120 (2009),
ArXiv:0804.3755.
- [22] M. Miyama and S. Kumano, Comput. Phys. Commun. **94**, 185 (1996),
hep-ph/9508246;
P.G. Ratcliffe, Phys. Rev. **D63**, 116004 (2001), hep-ph/0012376;
C. Pascaud and F. Zomer, hep-ph/0104013 (2001);
A. Cafarella and C. Coriano, Comput. Phys. Commun. **160**, 213 (2004),
hep-ph/0311313;
A. Cafarella, C. Coriano and M. Guzzi, Comput. Phys. Commun. **179**, 665 (2008),
ArXiv:0803.0462.
- [23] C. de Boor, ‘A Practical Guide to Splines’, Applied Mathematical Sciences 27,
Springer-Verlag New York Inc. (1978);
L.L. Schumaker, ‘Spline Functions: Basic Theory’, Krieger Publishing Company,
Malabar Florida (1993);
R. Kress, ‘Numerical Analysis’, Springer-Verlag New York Inc. (1998).
- [24] E. Eichten et al., Rev. Mod. Phys. **56**, 579 (1984).
- [25] G. Salam and A. Vogt in the QCD/SM working group report of the workshop
‘Physics at TEV Colliders’, Les Houches, May 2001, FERMILAB-CONF-02-410,
hep-ph/0204316.
- [26] S. Alekhin et al., in Proc. workshop ‘HERA and the LHC’ Part A, H. Jung and
A. De Roeck eds., DESY-PROC-2005-01, CERN-2005-014, hep-ph/0601012, pp.
119–159 (2006).
- [27] R.S. Thorne and W.K. Tung, in Proc. workshop ‘HERA and the LHC’, H. Jung
and A. De Roeck eds., DESY-PROC-2009-02, arXiv:0903.3861, pp. 332–351 (2009).

- [28] D. Roberts, ‘The Structure of the Proton’, Cambridge University Press (1990);
U.F. Katz, ‘Deep Inelastic Positron-Proton Scattering in the High-Momentum-Transfer-Regime of Hera’, Springer Tracts in Modern Physics (2000);
A.M. Cooper-Sarkar, R.C.E. Devenish and A. De Roeck, Int. J. Mod. Phys. **A13**, 3385 (1998), hep-ph/9712301.
- [29] W.L. van Neerven and E.B. Zijlstra, Phys. Lett. **B272**, 127 (1991).
- [30] E.B. Zijlstra and W.L. van Neerven, Phys. Lett. **B273**, 476 (1991).
- [31] E.B. Zijlstra and W.L. van Neerven, Phys. Lett. **B297**, 377 (1992).
- [32] J. Sanchez Guillen et al., Nucl. Phys. **B353**, 337 (1991).
- [33] W.L. van Neerven and A. Vogt, Nucl. Phys. **B568**, 263 (2000), hep-ph/9907472.
- [34] W.L. van Neerven and A. Vogt, Nucl. Phys. **B588**, 345 (2000), hep-ph/0006154.
- [35] S. Moch, J.A.M. Vermaseren and A. Vogt, Phys. Lett. **B606**, 123 (2005), hep-ph/0411112.
- [36] E. Laenen, private communication.

Index

- B-splines, 16–17
- backward evolution, 22–23, 31
- beta-functions, 6
- Bjorken- x variable, 8

- convolution integrals in QCDNUM, 18, 42
- convolution weights, 18
- custom evolution, 58–60

- DGLAP evolution equations, 7
- discontinuities in α_s evolution, 7, 13, 27
- discontinuities in pdf evolution, 13–14

- e^\pm basis pdfs, 11
- evolution of α_s , 6–7
- evolution of heavy quark pdfs, 14

- factorisation scale μ_F^2 , 8
- factorisation scale dependence, 65
- F_L' structure function, 65
- FFNS fixed flavour number scheme, 12
- flavour thresholds
 - on the factorisation scale, 9
 - on the renormalisation scale, 7, 9, 13
- forward substitution, 63–64
- Fourier convolution, 18
- fragmentation functions, *see* time-like evolution

- Gauss quadrature in QCDNUM, 44
 - accuracy parameter of, 31
- generalised mass (GM) schemes, 42

- HQSTF package, 69–71

- indexing of e^\pm basis, 41
- indexing of q, \bar{q} basis, 38
- interpolation mesh, 22

- ket notation, 10

- MBUTIL package, 24
- Mellin convolution, 8
- mesh point, *see* interpolation mesh
- $\overline{\text{MS}}$ scheme, 5
- multiple convolution, 18, 19, 47, 56, 57
- multiple evolution grid, 22, 27–29, 32

- non-singlet evolution, 9
- non-singlet quark density, 9, 10
- null value, 31
- number of active flavours n_f , 6
 - value at threshold, 9, 27, 52, 57

- parton density function, pdf, 6
 - number/momentum density, 8
- parton luminosity, 19
- pdf type, pdf set, 35, 39, 40, 58
- PDG convention, 26
- PEGASUS program, 14, 27
- polarised evolution, 9

- QCDNUM program
 - example job, 24–27
 - execution speed, 29
 - history, 5
 - list of subroutines, 30, 44, 50, 54
 - parameters in `qcdnum.inc`, 24
 - program hang-up, 29
 - web site, 24
- QCDNUM convolution engine
 - booktab, 44
 - copywgt, 47
 - efromqq, 51
 - fcrossf, 51
 - fcrossk, 50
 - getalfn, 52
 - getwpar, 49
 - idspfuns, 47
 - makewrs, 46
 - makewta, 45
 - makewtb, 46
 - makewtd, 46
 - makewtx, 46
 - nflavor, 51
 - qqfrome, 51
 - scalewt, 47
 - setwpar, 49
 - stfunxq, 52
 - tabdump, 49
 - tabread, 49
 - wcrossw, 47
 - wtimesf, 48

QCDNUM fast convolutions

fastclr, 54
 fastcpy, 58
 fastepm, 54
 fastfxf, 57
 fastfxk, 56
 fastfxq, 58
 fastini, 54
 fastkin, 57
 fastsns, 55
 fastsum, 55

QCDNUM routines

asfunc, 37
 chkpdf, 40
 dmpwgt, 35
 evolfg, 38
 ffromr, 37
 fillwc, 58
 fillwt, 35
 fpdfij, 41
 fpdfxq, 41
 fsnsij, 41
 fsnsxq, 41
 fsplne, 41
 fsumij, 41
 fsumxq, 41
 fvalij, 41
 fvalxq, 40
 getabr, 37
 getalf, 37
 getint, 31
 getord, 36
 getthr, 37
 getval, 31
 gqcopy, 34
 gqmake, 33
 grpars, 34
 gxcopy, 34
 gxmake, 32
 iqfrmq, 34
 ixfrmx, 33
 nwused, 36
 pdfinp, 39
 qcinit, 31
 qfrmiq, 34
 qqatiq, 34
 readwt, 36

rfromf, 37
 setabr, 37
 setalf, 37
 setint, 31
 setlun, 31
 setord, 36
 setthr, 37
 setval, 31
 splchk, 41
 xfrmix, 33
 xxatix, 33

renormalisation scale μ_R^2 , 6
 renormalisation scale dependence
 of pdfs, 10
 of structure functions, 65, 69
 rescaling variable χ , 43
 scale parameter Λ , 7
 singlet quark density, 8
 singlet-gluon evolution, 8
 singlet/non-singlet basis e^\pm , 11
 spline interpolation, 15–17, 41
 spline oscillation, 16, 22, 41
 splitting functions
 at leading order, 62
 perturbative expansion of, 9
 singularities in, 43, 62–63
 symmetries in, 8
 structure functions
 heavy flavour contributions, 68–69
 zero-mass structure functions, 64–66
 sum rule integrals, 26
 time-like evolution, 9, 62
 Toeplitz matrix, 19
 truncation prescription, 10, 52, 65
 t -variable, 18
 types of weight table, 43
 un-polarised evolution, 8
 VFNS variable flavour number scheme, 12
 weights, *see* convolution weights
 Wilson coefficient, 64
 y -variable, 18
 ZMSTF package, 66–68